DOCTORAL THESIS

Domain Specialisation and Applications of Model-Based Testing

Pari Salas, Percy

Award date:
2010

Domain Specialisation and Applications of Model-Based Testing

Percy Antonio Pari Salas

BEng (UCSM), MSc (USP)

A dissertation submitted in fulfilment of the requirements of the degree of Doctor of Philosophy for the School of Information Technology, Bond University.

March 2010

Typeset in LaTeX 2_ε

Statement of Originality

The work presented in this thesis is, to the best of my knowledge and belief, original, except where acknowledged in the text. I hereby declare that I have not submitted this material either in whole or in part, for a degree at this or any other university.

Percy Antonio Pari Salas

Date: March 31st, 2010

Submitted for examination: November 2009

Approved for the degree of Doctor of Philosophy: March 2010

Abstract

Software testing, one of the most important methods for quality assurance, has become too expensive and error prone for complex modern software systems. Test automation aims to reduce the costs of software testing and to improve its reliability. Despite advances in test automation, there are some domains for which automation seems to be difficult, for example, testing software to reveal the presence of security vulnerabilities, testing for conformance to security properties that traverse several functionalities of an application such as privacy policies, and testing asynchronous concurrent systems.

Although there are research works that aim to solve the problems of test automation for these domains, there is still a gap between the practice and the state of the art. These works describe specific approaches that deal with particular problems, generally under restricted conditions. Nevertheless, individually, they have not made noticeable impact on the practice in test automation for these domains. Therefore, there is a need for an integrated framework that binds specific approaches together in order to provide more complete solutions. It is also important for this framework to show how current test automation efforts, tools and frameworks, can be reused. This thesis addresses this need by describing a general model-based testing framework and its specialisation for the testing domains of security vulnerabilities, privacy policies and asynchronous systems.

The main characteristic of the general framework resides in the separation between behavioural (control) and data generation specifications. This framework is defined on the basis of labelled transition systems and context free grammars. Labelled transition systems allow behavioural models to be kept simple and tractable while extended context free grammars allow for the generation of data values that later will be used in the execution of test cases. The extended grammars in the data generation models contain a representation of the global state of the system, which allows for example, the history of the execution of test cases to influence the generation of subsequent data values.

Besides the general pattern described in the behavioural and data generation models, each specialised testing domain requires models that represent particular characteristics of the

system and the testing objectives for that domain. Vulnerability testing requires a model that describes the properties of a system that make it vulnerable and another one that describes what are considered to be the malicious intentions of an attacker. Privacy policies testing, requires the addition of a model that describes the conditions under which the execution of a defined operation is restricted or permitted. An important characteristic of the privacy policies described in this thesis is that they include the concept of obligations, this is, actions that require to be performed before the execution of the test case is considered successful. This framework considers test cases that fulfil bounded obligations.

In testing asynchronous systems, this thesis focuses in a defined subclass of systems in which actions can be partitioned into controllable and observable actions where execution of controllable actions is decided by the testing framework and observable actions designate the response of the system to the controllable stimuli. Different from other approaches for asynchronous systems, this thesis uses sets instead of queues to keep tracking of expected observable responses. This allows the present approach to deal with imperfect communication channels and with delays and loss of information, where the order of the observations is not important.

The practical applicability of the approaches presented in this thesis is demonstrated in several case studies from various domain applications, namely web-based applications, financial exchange protocols and operating systems. Particularly, the case study on operating systems demonstrates the integration of the general approach with an existing testing framework. This case study describes advantages, disadvantages and trade-offs of such integration.

Acknowledgements

I would like to start by acknowledging and thanking Dr. Padmanabhan Krishnan, my supervisor, for his thorough guidance, mentoring, patience and support throughout the development of this thesis.

I would also like to thank KJRoss and Associates and Smart Testing Technologies for funding my research. Particular mention deserves Dr. Kelvin Ross for many useful discussions and suggestions for the research project, and mainly for believing in me.

Many thanks go also to Bond University for providing the necessary support to undertake many research tasks, especially for supporting my attendance to various conferences and providing economic support during the writing of this thesis.

A general thank you goes to my colleagues and friends within the School of Information Technology and the Business Faculty at Bond University for making my time in the school a mostly enjoyable experience. However, particular thanks go to: Shane Bracher and James Larkin for countless chats, discussions and games in our shared office, for introducing me to the school, to bridge and to cricket; to Adrian Gepp for an enjoyable time sharing the same house, for many useful discussions and for proof reading some parts of this thesis; to Emma Chávez and Pedro Gómez for their unconditional friendship, the many chats in Spanish and for proof reading the early drafts of some chapters of this thesis.

Last but not least, I would like to thank my family, especially my parents, Nilda and Alfonso (RIP), for everything, and my wife Rita and my daughter Sarah, my two treasures on Earth, for so many wonderful moments, and for their unconditional love, support and patience.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation and statement of problems

In today's world, software is involved in a wide range of our daily activities. Computers (and their software) are considered to be some of the most useful tools for people to communicate, find information and to buy goods and services. With the fast and relatively inexpensive data transfer that the internet provides, and with the ever growing processing capabilities of today's computers, it makes sense to put them to use in order to try to earn an income or just to make life easier.

As we have become increasingly dependent on software systems, the quality of software has also become increasingly important. Numerous methods have been developed to improve and assure software quality. Software testing is one of the earliest [46] and among the most important methods used in industry for quality assurance.

Software testing has become a complex process for modern systems. Test automation is an attempt at solving the problems of software testing in terms of cost and reliability. Model-based testing is one approach that aims to support further and, if possible, to enable the full automation of software testing. However, test automation and the application of model-based testing approaches have developed in specific application domains while still presenting difficulties in others. The following sections of this thesis discuss software testing, test automation and model-based testing, the relationships between them and the problems

of their current approaches in specific application domains.

1.1.1 Software testing

In simple terms, software testing is an activity that amounts to observing the execution of a software system in order to validate whether it behaves as intended and to identify potential malfunctions [15]. Nevertheless, software testing embraces a variety of activities, techniques and actors, and its own complexity grows along the growing pervasiveness, criticality and complexity of the software systems.

Novel properties of modern computer programs make their testing difficult. Most of current software systems, for example, airline reservation systems, financial transaction systems and computer operating systems, are interactive and concurrent [127].

Interactive systems interact with an external environment that they do not control. For these kinds of systems, security has become an extremely important issue in software development. Interactive systems run continuously, reading inputs all the time and reacting to them, for instance, by sending outputs. The continuous trend towards distributed and mobile systems pervading everyday life and communicating through increasingly interconnected networks, poses increasing security risks for interactive systems because they are being exposed to hostile and malicious environments.

In concurrent software systems that communicate asynchronously, it may be impossible to tell beforehand in which order the operations in a program will be executed. At a higher level of abstraction, observable events that occur during the execution can be interleaved in many different ways. However, predicting the order of these events is impossible because the speed of execution may vary in different contexts at different times. In concurrent software, there may be errors which can be detected only if a certain interleaving of events takes place.

1.1.2 Test automation

In software development practice, testing accounts for as much as 50% of total development efforts and the increasing complexity of software systems makes software testing an error prone task [13]. In general, testing is a difficult, expensive, time-consuming and labour-intensive

process. Moreover, testing is (should be) repeated each time a system is modified. Hence, software testing is an ideal candidate for automation. Automation may help in making the testing process faster, less susceptible to errors and more reproducible [116].

Test automation has developed along time and has evolved from record and playback testing frameworks to data-driven and keyword-driven frameworks, resulting in benefits of scalability, better performance and testing and increased productivity [3]. Each of the automation approaches has its own advantages and disadvantages. A record and playback approach, for example, records a set of test activities and then plays them back repeatedly in order to carry out the testing process. It speeds up the testing process as the process can be repeated a number of times once the testing steps have being recorded appropriately in the testing scripts. However, usually the scripts used for this purpose contain hard coded values which must be changed if small changes are made to the application. This adds the overhead of updating the script. In the worst case scenario, all the tests must be re-recorded if complex changes are made to the application being tested.

The keyword-driven approach, also known as *action word* approach [26], is becoming the preferred method in industry test automation forums. It has better reusability than record and playback and data-driven methods [64]. However, from a practitioner's point of view [3], it requires more effort to be implemented initially, is more time consuming and requires, for testing application specific functions, proficiency in a scripting language.

An important amount of research effort has been allocated to study the automatic generation of test cases, including test oracles. Model-based testing is one technique that has received substantial attention [99, 39]. Advances in the theories of software testing enable further automation of the testing process, especially by exploiting the developments in model-based testing [15]. However, in general, the current practice of software test automation in industry is mostly limited to recording manual testing activities and replaying them as regression testing scripts [25].

1.1.3 Model-based testing

The general idea of model-based testing is as follows. An explicit behavioural model encodes the intended behaviour of an implementation called the system under test (SUT). Traces of the model are selected and these traces constitute test cases for the SUT. Models, represented as state charts, labelled transition systems or UML models, are more abstract than the SUT. Additional components perform the concretisation of the test cases to make them executable against the SUT [103].

Robinson [108] shows the differences between manual testing, automated testing with scripts and random testing with keywords, although Robinson does not use the term keywords, and compares them with model-based testing. According to Robinson, the key feature of the model-based testing approach is that it understands the application and knows what the application is supposed to do. Then, it generates test sequences dynamically and can detect when the application is not functioning properly.

Robinson [108] is not completely fair with the keyword-driven approach. This approach does not necessarily execute the "keywords" in a exclusively random fashion. Heuristics can be applied to guide the execution. Moreover, one can consider keyword-driven testing as a precursor to the model-based approach. In a way, keyword-driven testing can be viewed as a piece of model-based testing in that it provides the components for the test execution engine needed for it [27]. Keyword-driven testing, however, doesn't deliver the highly leveraged "automated test design" output that model-based testing provides.

Model-based testing is a natural approach if one considers that in any testing approach, automated or manual, the intended behaviour of the system is implicitly represented in a tester's mind [103]. The main contribution of model-based testing resides in representing this model explicitly. Additionally, model-based testing supports further automation by representing the system's model in a way that it can be read and understood by a machine.

Model-based testing provides important benefits. First, a model serves as a unifying point of reference that all teams and individuals involved in the development process can share, reuse, and benefit from. For example, confusion as to whether the system under test needs to satisfy a particular requirement can be resolved by examining the model. Secondly, the most

popular models have a substantial and rich theoretical background that makes numerous tasks such as generating large suites of test cases easy to automate. For example, graph theory readily solves automated test generation for finite state machine models [42].

Additional benefits of model-based testing are the possibility of finding more bugs and a lower maintenance effort than traditional test automation. More bugs can be found because an arbitrary number of different test cases can be generated from a single model. The maintenance is reduced to a few changes in the model where, for traditional automation, it would imply changes in hundreds or thousands of test cases.

Automated execution of test cases in model-based testing also brings benefits. It makes it possible to run a large number of generated test cases. It also enables a so called online model-based testing approach. In online testing, a test is generated simultaneously as it is being executed. This makes it possible to modify the course of a test execution according to the observed responses of the SUT.

1.1.4 Problems of current automation

Most test automation approaches have failed to support testing practitioners in the domain of security testing [120], while testing concurrent software is notoriously difficult due to problems with non-determinism and synchronisation [112].

Software security is defined as the absence of properties and features (commonly referred to as *vulnerabilities*) that pose a risk to the operator of the software or third parties if they are exploited with malicious intent [120]. Although the presence of security features such as cryptography, strong authentication techniques and access control plays an important role in software security, it is the presence of vulnerabilities that poses most of the security risks [101]. Most test automation approaches are able to test the functionality of the security features, however, they are not adequate to identify security vulnerabilities. Vulnerabilities are often subtle and hard to detect [129] and testing to reveal them inside an implementation requires a different approach [115].

Security vulnerability testing is motivated by understanding and simulating a hostile and malicious environment where an intelligent adversary is bent on breaking the system [101].

It is usually associated with *penetration testing* and *red-teaming*. Current *penetration testing* refers to executing a suite of scripted tests that represent known exploits [115]. One of the main problems of penetration testing as an assurance technique is that, usually, it depends on the tester's skills, and the knowledge and documentation of procedures, assumptions and requirements is poor or non-existent [8].

Additionally, testing for security related properties, such as privacy properties, has only been addressed in literature as functional testing for specialised software components such as *policy decision points* [81, 78, 80, 79], and inside relational databases [1, 29]. Nevertheless, these properties traverse several functionalities and issues related to them rarely appear within these specialised components. Therefore, there is a need for testing these security properties within all relevant functionalities of the system.

Testing and validation of asynchronous systems have to address different characteristics. Many of these characteristics, such as local non-determinism and communication delays, have been addressed theoretically [31, 18]. However, from a practitioner's point of view, these solutions are not always available. In particular, automated testing tools need to deal with practical implementation challenges. For example, perfect communication channels (without losses or delays) used in the theory are not present in real systems. Therefore, automated testing approaches need to handle systems with imperfect channels. In the same way, if some subsystems rely on external choices, these approaches need to provide mechanisms to handle non-determinism.

A shift to model-based testing could help to alleviate the problems with testing security vulnerabilities and security related properties as well as being able to help to address testing of asynchronous systems. However, it seems that the leap from traditional scripted testing to model-based testing is as hard as moving from manual to automatic test execution [51]. In practice, this shift does not occur because of a number of reasons:

- There is a lack of general knowledge on how to model (or represent) security vulnerabilities [120] and how to combine the representation of other security properties, such as privacy policies, with traditional behavioural models.

- Literature suggests that testers are required to know different forms of state machines, formal languages, and automata theory, probably one different formalism for each type of property or vulnerability [42, 51].

- Models themselves have also some drawbacks. The biggest one of those is the explosion of state-space needed. Even a simple application can contain so many states that the maintenance of the model becomes a difficult and tedious task.

- The necessary concretisation of test sequences derived from a model to be able to execute the test on the implementation is often neglected in model-based testing literature [129]. Therefore, practitioners have a constrained perception of model-based testing capabilities.

1.2 Aims and research question

The overall aim of this thesis is to investigate the characteristics of a model-based framework for the automated testing of, mainly, security vulnerabilities and security related properties such as privacy properties. It is also an aim to make this framework general enough to be applied in testing software systems in other domains. With focus on software assurance this framework needs to solve the problem of representing and documenting existing software vulnerabilities. Discovering new kinds of vulnerabilities in software systems will be out of the scope of this framework.

In a more detailed manner, the framework developed in this thesis should present the following features:

- provide an integrated modelling approach that supports the definition of the system's properties that characterise the presence of defined security vulnerabilities, possible ways of exploiting these vulnerabilities and the effects that a successful exploit has on the software behaviour;

- provide a way of integrating current descriptions of security properties, such as privacy policies, into this modelling approach;

- target the concretisation of test sequences for automated execution;
- facilitate the testing of systems in domains other than software security, such as asynchronous systems and operating systems; and
- maintain the advantages of data-driven and keyword-driven approaches and facilitate integration with existing test automation tools and approaches.

In summary, the research question that this thesis addresses is which features of model-based testing frameworks, e.g., type of models and definition of test objectives, support the automation of the testing process for security related properties such as security vulnerabilities and preservation of privacy? Do these features support the testing process for other non-related domains, e.g., asynchronous systems, and support the reuse of existing artefacts and tools in current testing frameworks?

1.3 Main contributions

The main contributions of this thesis are summarised as follows.

- The present thesis describes a general model-based testing framework that emphasizes the separation between behavioural models and data generation models. This separation makes it possible to keep models tractable while it also enables the generation of concrete test cases that include necessary data values to be used in their execution against the SUT. The generic specification of the models abstracts this approach from idiosyncrasies of more specific languages and makes it portable between domain-specific languages and tools. Behavioural models are described using labelled transition systems whereas data generation models are described in a general fashion using extended *context free grammars*.
- This thesis also describes how the general model is specialised for generating test cases on specific security testing domains, namely, security vulnerabilities testing and privacy policies testing. For each testing domain, behavioural and data generation models

become a general pattern of the system's specification while specific models represent special characteristics and the testing objectives for the particular domain.

The thesis establishes that testing to reveal the presence of vulnerabilities requires the modelling of the properties of an implementation that lead to a particular vulnerability (the implementation model) as well as the intentions and knowledge of a malicious user. Under the assumption that the behavioural model is correct, this thesis defines that a vulnerability is product of a *faulty* relationship between the behavioural model and the implementation model. The thesis calls *faulty context* the representation of this relationship. Thus, following a fault-based approach, this thesis defines in which contexts vulnerabilities can be present and exploited.

For the privacy policies domain, this thesis presents a general structured way of representing policies, particularly privacy policies. This thesis addresses the fact that (privacy) policies are traversal to the applications in the sense that they are applied to every occurrence or execution of a defined action, subject, of course, to the evaluation of a triggering condition. The privacy policies considered in this thesis include the concept of *obligations* and address their testing. The thesis also addresses the fact that on the presence of a contracted obligation the test case can only be evaluated after a second condition (linked to the obligation) has been triggered. In this case, the thesis makes an important assumption that is that unbounded obligations do not exist. The thesis provides a specific algorithm to drive the generation of test cases for this particular domain.

- A general framework should be applicable to different domains. This thesis demonstrates that this framework is applicable in domains different from security testing. Domains such as asynchronous systems testing and operating systems testing, provide interesting cases for studying the applicability of the framework.

  For the domain of asynchronous systems the thesis provides specific algorithms to drive the generation and execution of test cases. The provided algorithms are capable of dealing with imperfect communication channels, that is, channels with losses and delays,

where the order in which actions are observed does not necessarily reflect the order of execution.

In dealing with operating systems, this thesis demonstrates the testing of device drivers. It explores two slightly different approaches of modelling device drivers functionalities and compares them in terms of the number of test scenarios that each one produces and the granularity of control over the operations executed.

- The thesis also demonstrates, via examples, that the presented approaches can be implemented using existing tools and can also be integrated with current testing frameworks. It presents the advantages, disadvantages and trade-offs of such integration focusing on the level of detail of the models.

1.4 Thesis outline

The following provides a brief outline of content, for all subsequent chapters in this thesis.

- Chapter 2 presents a general overview of software testing and test automation in the domains of security testing (vulnerabilities and privacy properties) and testing of concurrent asynchronous systems; with a particular focus on the model-based testing approach. It also describes commonly used modelling formalisms that serve as a basis for the models presented in Chapters 3 and 4. The aim of this chapter is to equip the reader with essential-knowledge that provides a point-of-reference for understanding the description of the testing framework presented in this thesis.

- Chapter 3 presents a general model-based framework for test automation that can be applied to a wide range of systems and testing domains.

- Chapter 4 describes specific details that allow the general framework to be specialised and applied to the testing domains of security vulnerabilities, privacy policies and asynchronous systems.

- Chapter 5 presents four case studies. In the first case study, the general framework is used in the domain of operating systems to demonstrate its practical applicability and

the reuse of current test automation frameworks. The second and third case studies demonstrate the applicability of the framework specialised for testing security vulnerabilities and privacy policies in the context of web-based applications. The fourth case study shows the framework, specialised for testing asynchronous systems, applied to the testing of the Financial Information eXchange (FIX) protocol.

- Finally, Chapter 6 presents the thesis conclusion - a brief summary of work carried-out, conclusion and directions for future research.

Chapter 2

Literature review

2.1 Preliminaries

The testing framework in this thesis is based on the general formalism of *labelled transition systems*. In this section, general concepts related to labelled transition systems are presented with the aim of providing the reader with the essential knowledge needed to understand the presentation of the framework described in this thesis.

2.1.1 Labelled transition systems

A labelled transition system is a structure consisting of states with transitions, labelled with actions, between them. The states model the system states; the labelled transitions model the actions that a system can perform.

Definition 1 *A labelled transition system (LTS) is a 4-tuple* $(Q, A, \rightarrow, q_0)$ *where*

- Q *is a non-empty set of states of the system,*
- A *is a set of action labels,*
- $\rightarrow$ *is the transition relation,* $\rightarrow \subseteq Q \times A \times Q$*, and*
- q_0 *represents the initial state of the system.*

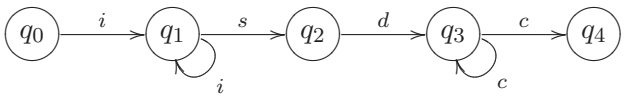


Figure 2.1: LTS model of a vending machine

The transition relation $\rightarrow$ denotes possible state changes; if $(q, a, q') \in \rightarrow$, one can say that the system can move from state q to q' by performing action a. As customary, $q \xrightarrow{a} q'$ is written instead of $(q, a, q') \in \rightarrow$. Transitions can be composed. Suppose that in state q' the system can perform action a', which means $q' \xrightarrow{a'} q''$. Then one can compose, $q \xrightarrow{a} q' \xrightarrow{a'} q''$, which is written as $q \xrightarrow{a \cdots a'} q''$. Composition of transitions can be generalised to $q_i \xrightarrow{a_1 \cdot a_2 \cdot \ldots \cdot a_n} q_j$ which expresses that the system, in state q_i, can perform the sequence of actions $a_1 \cdot a_2 \cdot \ldots \cdot a_n$, and may end in state q_j. As it is well pointed out by Tretmans [119], the use of may is relevant because of possible non-determinism. That is, a composed transition can exist such that with the same sequence of actions, the system will end in a state other than q_j, e.g. $q_i \xrightarrow{a_1 \cdot a_2 \cdot \ldots \cdot a_n} q_k$.

Consider the graph in Figure 2.1. It is a pictorial representation of a LTS that models the behaviour of a vending machine. That is, it models a machine that accepts some coins as payment for an item, lets the user choose the desired item and returns some coins as change if needed. To improve readability, consider actions i for inserting coins, s for processing the user selection, d for delivering the item, and c for giving back the change. The non-determinism referred to in the previous paragraph is present in this simple model. Consider the sequence of actions $\sigma = i \cdot i \cdot s \cdot d \cdot c$. One can derive two composed transitions from this sequence, $q_0 \xrightarrow{\sigma} q_4$ and $q_0 \xrightarrow{\sigma} q_3$. The first transition means that the machine has delivered the item and all the change due; the second transition represents the case where the machine has still some change to give back to the user.

A labelled transition system contains information about all possible behaviours of a system. However, in software testing one usually needs to reason about a particular behaviour or subset of behaviours. To achieve this, the following concepts associated with a LTS need to be introduced.

Definition 2 (Path) *Given a LTS* $M = (Q, A, \rightarrow, q_0)$*, a* path *in* M *is a sequence* $\pi = q_0 a_0 q_1 \cdot \ldots$ *such that for all* i *we have* $q \xrightarrow{a} q'$*. We denote with* paths(q) *the set of paths*

starting in q. We use paths(M) for paths(q_0). With Paths(q,q_n) *one denotes the set of paths starting in q and ending in q_n. As a notational convention, write $q \longrightarrow q_n$, if* paths(q,q_n) *is not empty and $q \longrightarrow$, if there exists a state q_n such that $q \longrightarrow q_n$.*

Definition 3 (Trace) *The trace α of a path π, denoted trace(π), is the sequence $\alpha = a_0 \cdot a_1 \cdot \ldots a_n$ of actions in A occurring in π. With traces(M) = $\{trace(\pi) | \pi \in paths(M)\}$ one denotes the set of traces in M. Analogously, traces(q) denotes the set of traces of all paths that start in state q. In case α is finite, $|\alpha|$ denotes the length of the trace α and the function* last(α) *returns the last action in α.*

Additionally, there are circumstances in which particular properties of a state need to be described. With this objective in sight, this thesis defines a *state predicate* as a Boolean-valued function built from system states. Formally, given a suitable universe of constants and function symbols with fixed semantics, the set of all possible state predicates is defined as $State\ predicates = \{sp \mid sp : Q \rightarrow Bool\}$. Given a state predicate sp and a state q, $sp(q)$ determines if sp holds in q.

2.1.2 Input-Output transition systems

A labelled transition system specifies the possible interactions that a system may have with its environment. These interactions are abstract, in the sense that they are only identified by a label; there is no notion of initiative or direction of the interaction. Usually the environment can also be modelled as a LTS. In this case, an interaction can occur if both, the system and its environment, are able to perform that interaction. That is, the system and its environment synchronise over common actions.

This paradigm of abstract interaction is sufficient for analysing and reasoning about a large number of applications [119]. However, many real world systems communicate with their environment in a different way. For these systems, there is a clear distinction between inputs and outputs, where inputs are actions initiated by the environment and outputs are actions initiated by the system. The literature assigns two important properties to these systems:

- outputs from the system are never refused by the environment, and
- inputs from the environment are never refused by the system.

The second one is the most important and is also known as the *input-enabled* property.

Input-output transition systems are a special kind of labelled transition systems that model this class of systems with inputs and outputs.

Definition 4 *An* input-output transition system (IOTS) *is a labelled transition system where the set A of actions is partitioned into disjoint subsets of input actions A_I and output actions A_O and where all input actions are enabled in any reachable state:*

$$\forall q' \cdot q \longrightarrow q' \ , \ \forall a \in A_I : q \xrightarrow{a}$$

Consider the set A of actions in the LTS of Figure 2.1. Then, consider disjoint subsets $A_I = \{i, s\}$ and $A_O = \{d, c\}$ such that $A = A_I \cup A_O$. Even under these considerations, the system in Figure 2.1 is not an IOTS because it lacks the input-enabled property. However, in a LTS there can be some states that refuse certain input actions, and thus one can opt for not showing them explicitly. Then, a labelled transition system can be seen as a partially specified input-output transition system. For example, in the vending machine example, it refuses any user selection before a coin has been inserted, and refuses additional coins to be inserted after the selection has been made.

There are two reasons for writing partial specifications. One is that it does not matter how implementations respond to unspecified input actions. The other is that the environment is assumed not to offer such inputs, so there is no need to specify them [54]. We believe the latter is usually the reason for not specifying certain input actions. Nevertheless, sometimes our assumptions are not highly accurate when it comes to real implementations. Thus, models need to be refined and specify how the system deals with all inputs.

One way of completing a partially specified IOTS is to add a special state to the system and to add transitions to this state for all non-specified inputs. This way assumes that the reason for under-specifying certain input actions is that one doesn't care or know what the

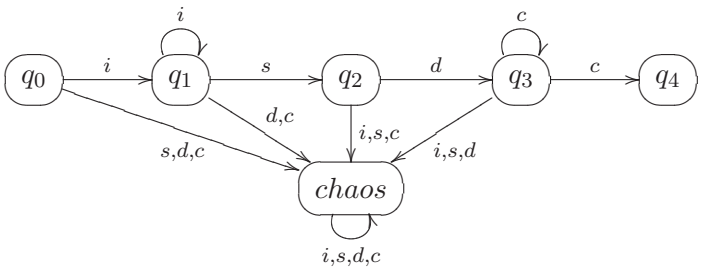


Figure 2.2: Demonic completed IOTS model of a vending machine

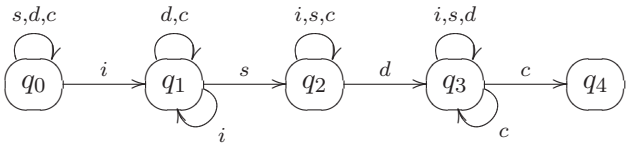


Figure 2.3: Angelic completed IOTS model of a vending machine

behaviour of the system is after those actions have been executed. This means that in an underspecified state – i.e., a state reached after an unspecified input action – every action from the label set is correct. This behaviour is known as *chaos* [62], and the method that completes the IOTS by introducing it, is called *demonic completion*. A characteristic of demonic completion is that the new state acts as a sink. That is, once the system reaches such a state it will never leave that state again. Figure 2.2 shows the demonic completed IOTS of the vending machine's LTS in Figure 2.1.

Another way of making systems input-enabled is, in every state q and for each non-specified input action a, to include a self-transition $q \xrightarrow{a} q$. This self-transition makes the system accept those inputs that were previously refused but without changing the system's state. This method is also referred to as *angelic completion*. In Figure 2.3 we show the angelic completed IOTS for our vending machine.

The input-enabled property of IOTS brings on another concept for one to deal with. Since all input actions are enabled in every state of the system, it is up to the environment to decide whether an input action will occur, or not. Conversely, since the environment is said not to refuse any output action, output actions depend only on the system to be executed. All this means that in a given state a system can choose between executing any output action or waiting for the environment to execute an input action. However, there will be, possibly, states

where no output actions are possible and the only alternative is to wait for the environment. Such a state where the system cannot autonomously proceed is called a *quiescent state*.

Definition 5 (Quiescent state) *Let* $M = (Q, A_I \cup A_O, \rightarrow, q_0)$ *be an IOTS. A state* $q \in Q$ *is* quiescent*, denoted by* $\delta(q)$ *if* $\forall \omega \in A_O : q \stackrel{\omega}{\not\longrightarrow}$.

Definition 6 (Quiescent traces) *The* quiescent traces *of a IOTS* M *are those traces that may lead to a quiescent state. This is,* $Qtraces(M) = \{\sigma \in A^* | \exists q' : q_0 \stackrel{\sigma}{\longrightarrow} q' \wedge \delta(q')\}$

The framework in this thesis does not mention explicitly the concept of quiescence but it is implicitly stated in the models of asynchronous systems when the tester defines the accepting states. In other words, quiescent states in asynchronous systems define the accepting states of an asynchronous model.

2.2 Software test automation

Test automation refers to the process of using computers to assist in the process of software testing. The main goal of test automation is to make the testing process faster, less susceptible to errors and more reproducible [116].

Test automation has developed along time and has evolved from record and playback testing frameworks to data-driven and keyword-driven frameworks [3]. Hayes [53] presents this evolution from the practitioner's perspective. According to Hayes, because of their cost of maintenance, pure record and playback tools were rapidly modified to include programming features giving place to scripting frameworks (also known as linear frameworks). The different approaches followed by test automation frameworks are summarised in Table 2.1 [3, 53, 68].

In the remaining part of this section, this thesis describes the keyword-driven framework in greater detail. As it was defined before, one of the aims of this thesis is to maintain and take advantage of the advances in the execution of test cases that current test automation frameworks present. Later, model-based testing is also described in greater detail. This thesis uses model-based testing with the aim of providing an integrated framework that addresses test case generation and execution.

Table 2.1: Test automation frameworks

Framework type	Description
Linear	Automated scripts contain all necessary components for the execution of the test inside their bodies. This approach lacks modularity and scripts hardly can be reused.
Functional decomposition	This is a framework that involves the creation of modular functions for executing fundamental actions that occur during test execution. These actions may then be called upon and executed independently of one another, thus allowing them to be reused by multiple tests within an automated test suite.
Data-driven	An enhancement to the functional approach in which most of the components to be executed still exist inside the body of the script. However, the data used in these scripts is typically stored in a file that is external to the script, which promotes script reusability.
Keyword-driven	A further enhancement to the functional approach. In this approach the logic of the execution is separated from the executable components themselves. These components remain inside the scripts' bodies but the logic is described by a set of keywords (functions at a much higher level of abstraction than in the functional approach). Tests are developed in a tabular format that is interpreted and executed by a driver script or utility.
Model-based	Often called "Intelligent frameworks", model-based frameworks go beyond creating automated tests that are executed by the tool. Provided with information about the application, in the form of state models, these frameworks generate and execute tests in a semi-intelligent, dynamic manner.

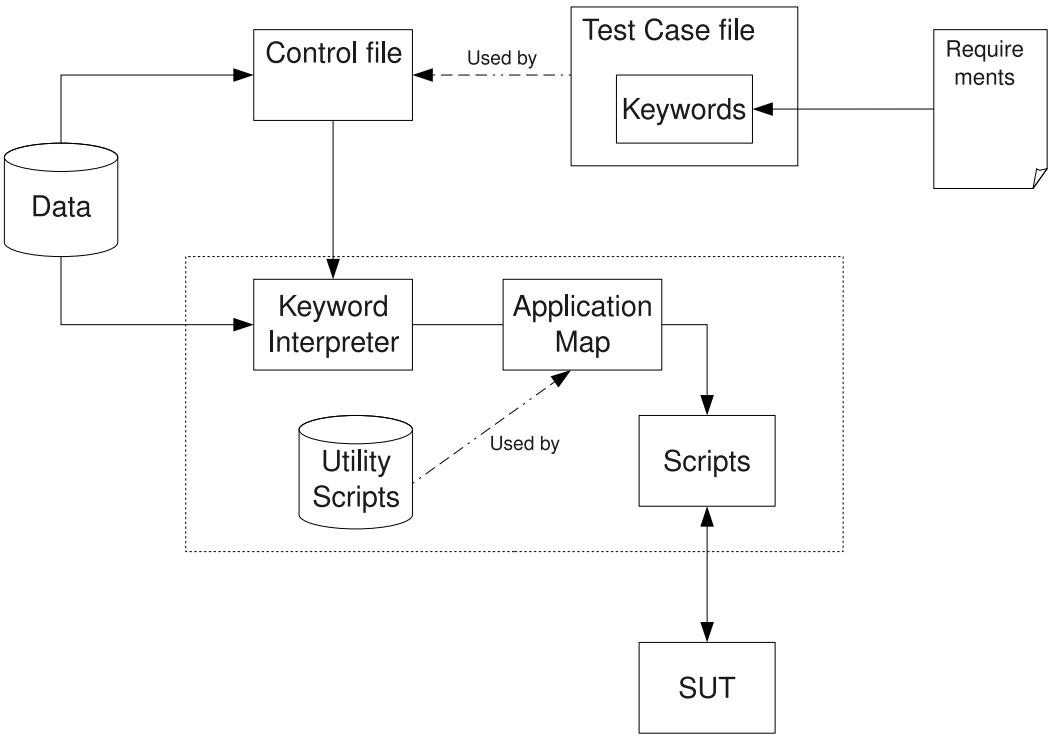


Figure 2.4: A general architecture for a keyword-driven testing framework

2.2.1 Action-words and keywords

The keyword-driven approach [43], also known as table-driven approach [52] and action-words [26], is a test automation approach that separates the scripts that execute automated test cases from their logic and test data. Keyword-driven testing is a logical extension to data-driven testing.

In a simplified way, a keyword-driven framework contains three elements: a library of keywords, a set of data tables and a set of test scripts [3]. A keyword is typically a verb or verb-like phrase that is associated with an external library function (the script) responsible for executing the application commands necessary to satisfy the objective of the test. The data tables provide the necessary data elements for the scripts to work. These elements are used and interact in a testing framework as depicted in Figure 2.4.

The main elements of the keyword-driven framework in Figure 2.4 are defined as follows:

Test Case file. Contains the detailed steps to be carried out for the execution of a test case. It is in the form of a table that contains columns with information for each step, such as *keyword*, *object name* and *parameter*.

Control file. Contains details of all the *test scenarios* to be automated. Control files are usually tables that contain the scenario ID, the path to the data repository, the path to specific *test case* files and a flag that defines if a defined test case will be executed or not. Usually the tester is able to select a specific scenario to execute based on turning on or off this flag.

Keyword Interpreter. Reads information from the test case and control files. In case parameters are defined using variables, it performs the matching between the variables and their values in the data repository.

Application map. Is one of the most critical components, which is used for mapping the system's operations and objects, from names human testers can recognize to a data format useful for the automation tool. For example, when testing a GUI, a naming convention can refer to each component in each window by using the type of object and the name of a property (e.g. button with label "login", text field with label "password"). Then use the Application Map to associate that name to the identification method needed by the automation tool to locate and properly manipulate the correct object in the window.

Scripts. Modular reusable routines or functions that perform generic tasks by calling the SUT's interface. The scripts (as shown in Figure 2.4) are particular instances of the *utility scripts*. The *utility scripts* are completely generic in that they are based in the type of object(s) and the operation to be performed. It is the *application map* that selects the *scripts* to be executed and defines the exact instance of the object(s) to be used during the execution of the script.

The two most important advantages of working with action words are probably readability and maintainability [26]. Tests are easy to read because all details needed for their execution

(like which buttons have to be pushed or at what location on the screen an outcome can be found) are hidden behind the action words. Moreover, the application map enables the scripts and keyword driven tests to have a single point of maintenance, on the task of identifying particular objects in the SUT's implementation. For example, if a new version of an application changes the title of the window or label of the components, it should not affect the test cases. The changes will require only a quick modification in one place–inside the Application Map. In this thesis we aim to maintain these advantages of the keyword-driven approach.

The action-words approach is fairly flexible and generic, and this makes it suitable for implementing an approach as specialized as model-based testing [27]. A first way of representing a model in the action-words framework is by using decision tables in the control file. With decision tables the execution of actions can be selective and repetitive. A table that contains different rows with specific data values for each can be used to select the action to be executed. Data values can be generated in a loop to trigger the execution of the correspondent action as many times as desired.

The relationship between models and the action-words approach is not limited to decision tables. Complete state machines can be represented in action-words. Buwalda [27] uses the terms *situation* and *move* to describe nothing more than states and transitions. Situations are explicitly given a name (tag) and moves describe, also explicitly, the "jump" from one situation tag to another.

There is no reason to limit the description of the model to the approaches cited above. The way in which a model is described depends on the skills of the testers. They need to understand the problem area and the model technique(s) involved [27]. Thus, by allowing the testers to describe models in different formalisms, one can reuse the implementation of the action-words (the script code) and fully link action-words with model-based testing.

2.3 Model-based Testing

Extensive research has been performed in the area of model-based testing. Annotated bibliographies and surveys at different points in time during the last decade have reported the

advances in this area [99, 122, 39, 60]. Some authors (see for example [76, 71, 96, 113]) refer to model-based testing also as specification-based testing. In this thesis the term *model-based testing* is preferred but the terms model and (formal) specification are used without distinction (unless explicitly stated the contrary). In any case, both terms refer to the introduction and use of software models (or formal specifications) into the testing process.

Formal software specifications have been incorporated into testing in several ways. They have been used as a mean of executing testing into the earlier phases of software development as well as a basis for test case generation. On one hand, by using executable specifications in evolutionary prototyping, the testing process can start much earlier and can therefore be more effective [104]. On the other hand, since formal specifications define mathematically the behaviour of a piece of software, complete test cases can be derived from them. The *de facto* meaning of model-based testing is related mainly to test case generation.

Model-based testing is a black-box testing technique that relies on explicit behavioural models that encode the intended behaviour of a system and possibly the behaviour of its environment. The idea that motivates this technique is that the existence of an artefact that explicitly defines the intended behaviour can help mitigate the problems that traditionally appear on the process of designing test cases (unstructured, not reproducible, not documented, among others). In a recent survey article, Utting et al. [122] define *model-based testing* as the automatable derivation of concrete test cases from abstract formal models, and their execution.

Based on the works of Hierons et al. [59] and Utting et al. [122] a generalised process of model-based testing is presented in the following section. This process contains four steps that refer to three main elements: models, the process of generating test cases and the process of executing these test cases. Issues related to these three elements are also discussed.

2.3.1 Process

A generic process of model-based testing usually involves four stages:

- **Stage 1.** Building an abstract model of the system under test. This is similar to the process of formally specifying the system, but the kind of specification/model needed

for test generation may be a little different to that needed for other purposes, such as proving correctness, or clarifying requirements. The level of abstraction of the model is also variable. Sometimes, abstraction refers to the fact that the model neglects or disregards certain functionality or quality attributes, such as security.

An initial validation of the model can be considered on this stage. However, Utting [121] does not consider it to be a crucial step when the case is test case generation. It is because if the model contains errors, the generated test cases will find and report them.

- **Stage 2.** Defining test selection criteria and abstract test case specifications. The selection criteria describe, possibly informally, a test suite. In general, it can relate to a given functionality (when it is based on requirements), to the structure of the model (state coverage, transition coverage, predicate coverage), to stochastic characterisations (randomness, user profiles) and also to a well-defined set of faults. A test case specification formalises the notion of the selection criteria in terms of the same language as the model, or maybe operations over the model.

- **Stage 3.** Generating a test suite. Given the model and a test case specification, an automatic tool is capable of deriving concrete test cases. In general, there are many test cases that satisfy the specification and generators pick some in a random way, by applying some heuristic like picking boundary-values, or by using elaborate searching algorithms to find them. However, in some rare cases, the set of test cases that satisfy a test case specification can also be empty. In such cases, either the domain of values used in the generation process is considered inadequate and another set of values is required, or, if all possible values have been considered, the model does not present the property represented by the test case specification.

- **Stage 4.** Executing test cases. Consider that a test case is composed of three distinguishable parts, input data, expected output (which includes data) and a control sequence. This control sequence defines the set of steps or operations required to realise the test cases. In executing a test case, the SUT runs following what is indicated in the control sequence using the input data as parameters and comparing the oracle (output

expected by the test case) with the actual results from the SUT. This comparison forms a verdict, which can take the outcomes of pass, fail and inconclusive. A test case passes when expected and actual output conform. It fails when expected and actual outputs do not conform, and it is inconclusive if no decision can be made.

It must be recalled that model and SUT reside at different levels of abstraction, and that these different levels must be bridged. Hence, the input part of the test cases could need to be adapted for execution on the SUT, and outputs could also need to be abstracted to match the oracle.

2.3.2 Models

According to Offutt [96], there are three main approaches inside specification-based testing: model-based, transition-based and property-based. Each one is different from the other on the type of notation they use for describing the specifications. Although presented here as different approaches, later this thesis avoids the dichotomy between model-based and property-based models.

Model-based specification languages, such as Z, B, VDM and OCL, attempt to provide formal specifications of the software based on mathematical models. These languages generally use pre- and post-conditions to describe the relation that exists between inputs and outputs of a system. Transition-based specifications describe software in terms of state transitions. Typical transition-based specifications define preconditions on transitions and triggering events. Preconditions are values that specific variables must have for the transition to be enabled. Triggering events are changes in variable values that cause the transition to be taken. These events "trigger" the change in state. Algebraic specification (or property-based) languages describe software by making formal statements, called axioms, about relationships among operations and the functions that operate on them. This kind of specification is also known as *functional specification*.

Other approaches have been used in specification-based testing. Most of them extend the previous ones by adding mechanisms to specify additional characteristics like time, concurrency and parallelism, and probabilities. Utting et al. [122] describe them as history-based,

operational and stochastic notations. History-based notations model a system by describing the allowable traces of its behaviour over the time. Various notions of time can be used, leading to different kinds of temporal logics. Formalisms like *message sequence charts* are also considered in this group. Operational notations describe a system as a collection of executable processes, executing in parallel. This group includes the process algebras, like CSP and CCS, and Petri nets. Stochastic notations describe a system using a probabilistic model of the events and input values. In general they are used to describe the system's environment rather than the system itself. As an example, Markov chains are used to model expected usage profiles so that the generated test cases exercise that usage profile [128].

In addition to selecting which approach should be used to describe the model(s) of the system, it is equally important to define *what* is going to be modelled. This is called the **model subject** [122]. A model can either describe the intended behaviour of the system or describe the possible behaviour of the environment. On one hand, the model of the system serves to define an oracle for its functionalities. Moreover, its structure can be exploited for the generation of the test cases. On the other hand, the model of the environment is used to restrict the possible inputs to the model of the system, acting then as a test selection criterion. Environment models can also be used to define exactly the *stimuli* that exert only certain "parts" of the system where the tester believes there is a major probability of finding problems.

Most often, for test case generation process, both models are used. Different combinations of the models serve different purposes. Figure 2.5 illustrates the possible combinations. The vertical axis shows how much of the behaviour of the system is modelled, while the horizontal axis shows how much of the environment is modelled. The shaded area shows all the possible combinations. In the following some combinations are discussed.

Extreme positions are marked with S,E and SE. The model S includes all the details about the system but says nothing about the environment. Then, the input space for the system is not constrained, and any behaviour of the environment is expected. Model E is the opposite, where full knowledge of the environment and nothing about the expected behaviour of the system is explicitly represented. This constrains the input space for the system but

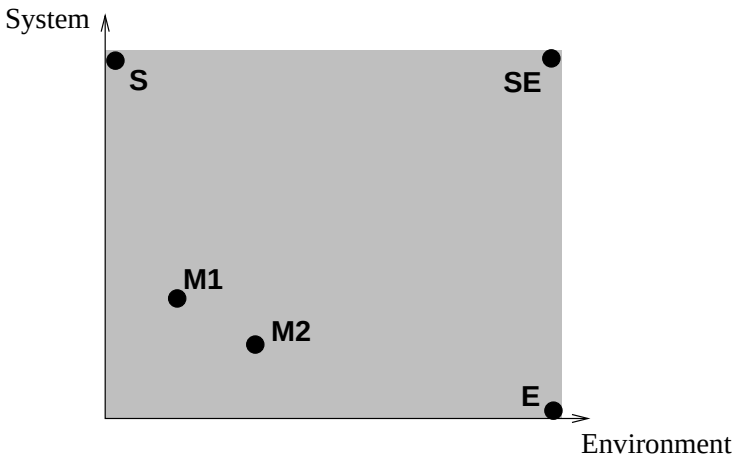


Figure 2.5: Model subjects of model-based testing (source: [122])

cannot be used as an oracle to verify its execution. Position SE is the most extreme case where everything about the system and the environment is modelled. This kind of model would be as complex as the implementation of the system (or even more). Thus, it is not practical. Models in positions $M1$ and $M2$ are the likely ones to be used in model-based testing. Those models combine certain aspects of both, the system and its environment, and simplify the understanding by hiding the irrelevant details. How much and which details are hidden depends on the level of abstraction of the models. It is important to note that the level of abstraction can be induced by the modelling language itself by not providing any means of modelling certain aspects, or can be defined by the modeller who explicitly disregards certain information.

2.3.3 Test case generation

From a well described set of models one can generate test cases. Two main issues have to be discussed when the test case generation takes place. First, it has to be defined *which* test cases are going to be considered. Secondly, it has to be defined *how* these test cases are going to be generated. The answers for *which* test cases are generated refers to what is called *test selection criteria* and the answer for *how* they are generated refers to the *test generation technology*.

A test selection criterion defines the characteristics that will be evaluated on a test case

for it to be considered relevant. Traditionally, three major criteria have been used in testing: structural coverage criteria, data coverage criteria and fault-based criteria. Other criteria such as ad-hoc test case specifications and random and stochastic criteria have also been used on a minor scale. The approach presented in this thesis emphasises the use of fault-based criteria. It is not that the other criteria are not important but, according to Morell [89], the fault-based approach is more adequate when the motivation is to demonstrate the absence of pre-specified properties (or faults).

Fault-based criteria. The fault-based criteria use an alternate way of showing program correctness. This alternate way is based on falsification, which means that it shows that the program is not incorrect. If a program has a limited potential for being incorrect, then a test set demonstrates correctness when it shows the potential is not realised. A fault-based criterion tries to specify incorrectness by defining potential faults for program constructs. The most popular criterion of this kind is mutation testing [50]. Initially mutation testing creates mutant programs from an original one by altering a single program construct, for example, replacing one logical operator with another. The rules that define which changes are valid or useful are called mutation operators. Subsequently, the mutants are executed with a defined test case and the execution results are compared with the execution results of the original program. If they are different then the test case distinguishes the original program from the mutant, hence it reveals the flaw inside the mutant program. A test case (or test suite) is as good as the percentage of mutant programs it distinguishes. Of course, even a test suite with a perfect mutation score (100% of mutants distigushed) is only guaranteed to be effective against those faults introduced into the mutant programs. This characteristic is common to all fault-based approaches.

The test generation technology refers to the technique that is used to actually construct the test cases. Again, different techniques have been explored by different research works. Manual derivation of test cases, for example, is always a possibility, however it does not take real advantage of the potential of automation of model-based testing and that could be considered a waste. In the following sections this thesis discusses briefly some automated

techniques used in test case generation, such as graph search algorithms, model checking, and constraint solving.

Dedicated graph search algorithms. There are limitless numbers of existing graph algorithms which can be applied to determine a test path, for example, the shortest round trip, depth first search and most likely paths. Even a random path algorithm that generates any sequence of transitions on the graph can be used for stress testing. However, a random path does not guarantee coverage of all states and transitions and this is not meaningful in a large testing scheme. One of the more effective and efficient graph algorithms used to satisfy coverage is the Chinese Postman algorithm [22]. The Chinese postman algorithm has been already applied successfully in protocol testing and behaviour testing especially when state charts are used as models. The interested reader is referred to the work of Petrenko [99] for a detailed bibliographic survey on test case derivation using this technique.

Model checking. The purpose of a model checker is to verify or falsify specified properties of a model. In order to use a model checker for test case generation, a set of properties (usually reachability properties) is generated and the model-checker is asked to verify the properties one by one. These properties are constructed in such a way that they fail for the given system specification, leading the model checker to produce a counter-example. The counter-example shows a valid sequence of states that any conforming implementation should follow. This sequence of states becomes a test case. The main challenge in generating these properties is to generate them in such a way that the set of counter-examples generated will be adequate to satisfy a previously defined coverage criterion.

Constraint satisfaction. Most of the technologies described before (except for random testing) have to satisfy some constraints. Thus, the problem of generating test cases from a formal specification can be represented as a Constraint Satisfaction Problem (CSP). A constraint satisfaction problem consists of a finite set of variables and a set of constraints. Each variable is associated with a set of possible values, known as its domain. A constraint is a relation defined on some subset of these variables and denotes valid combinations of their values. A solution to a constraint satisfaction problem is an assignment of a value to each variable from its domain, such that all the constraints are satisfied. Formally, the conjunction

of these constraints forms a predicate for which a solution should be found.

When linked to the testing terminology, the predicate for which a solution is searched is the *test case specification*. Domains for the variables of the CSP are directly related to the data type defined for the variables in the test case specification. Then, a *concrete test case* is the set of state, input and output variables, with a value of their domains assigned to them.

A constraint solver implements an algorithm for solving well-formed constraints within a CSP in accordance with a constraint theory. The constraint theory $\mathcal{CT}$ defines the semantics of a constraint system and is composed of a set of transformation rules over one or more specific domains. The syntax of well-formed constraints is defined by the set of allowed constraints $\mathcal{C}$. At the heart of any constraint solver there is a search method that looks among many combinations for a valid solution for a given predicate. Search methods for solving constraints include alternating variable, simulated annealing, genetic algorithms, iterative relaxation and different heuristics. McMinn [86] has reported the use of genetic algorithms, simulated annealing, and evolutionary algorithms for test case generation. The author has developed previously [2] a constraint solver that uses a branch-and-bound tree to search over partially ordered types with the aim of generating test data for unit testing.

2.3.4 Test case execution

The last step in the testing process is the test case execution (and evaluation). Test execution refers to the activity of running the *system under test* (SUT) using the generated test case inputs and comparing the results of this execution with the generated expected outputs. Two approaches for test case execution are relevant to us: the on-line and the off-line approach.

The on-line approach mixes test case generation and execution. Thus, some part of a test sequence is generated and then applied on a SUT execution. Then, the test generation algorithm generates another part of the test sequence and executes the SUT with this (partial) test, repeating this process several times. The advantage of this approach is that the test case generation algorithm can react to the actual outputs of the SUT. This is especially useful under the presence of non-determinism in the SUT and to avoid problems of state explosion when specifications on a low level of abstraction are used. This is also known as the *on-the-fly*

approach.

The off-line approach completely separates test case generation from execution. Thus, all the test cases are generated strictly before they are run. The advantages of this approach include the fact that generated test cases can be applied using existing test management tools and a set of test cases can be generated once but run many times. The last fact is especially useful in regression testing and when the generation process is slower than the test execution.

2.4 Software security

Software security is the idea of engineering software so that it continues to function correctly under malicious attack [85, 120]. The first books and academic classes on this topic appeared in 2001 [5, 125], although some previous research work can be identified in the area of software vulnerabilities [106, 41, 40]. In the last decade, a number of other works have provided a philosophical underpinning for software security and have discussed particular technical issues. This thesis focuses on technical issues related with software testing and the use of models to support it.

Security testing can generally be classified into *security functional testing* and *security vulnerability testing* [101, 85, 82]. Security functional testing involves testing the product or implementation for conformance to the security function specifications, for example, the specification of an encryption module, as well as for the underlying security model, generally described as a security policy, for example, an access control policy or a privacy policy. The conformance criteria state the conditions necessary for the product to exhibit the desired security behaviour or satisfy a security property. In other words, security functional testing involves what the product should do. Security vulnerability testing on the other hand is concerned with the identification of flaws in design or implementation that may be exploited to subvert the security behaviour which has been made possible by the correct implementation of the security functions. In other words, security vulnerability testing involves testing the product for what it *should not do*.

Model building is a standard practice in software engineering. The construction of models

during the systems design improves the quality of the resulting systems by providing foundation for early analysis and fault detection. Testing activities have found a useful partner in formal modelling, giving raise to *model-based testing*. It seems security testing is (slowly) joining this approach. Model building is also carried out in security and policy specifications. In fact, a model of the system will provide a clear figure of what must be protected and which interaction points (with the environment and potential malicious entities) need to be addressed. However, as Basin et al. point out in [12], there are two gaps in the integration of system and security modelling into the overall development process: 1) security and system models are typically disjoint and described in different ways (languages), and 2) even when security properties are considered in early models and security mechanisms are implemented into the system, there is hardly a mapping between the two. This thesis addresses this problems by defining, when possible, a general way of representing security and system models and, when it is not possible, by defining an algorithmic way of mapping those models.

Diverse testing techniques, including model-based strategies, can be applied with success for (functional) testing of security modules such as intrusion detection systems and modules for enforcing access control methods (see [72] for more examples). However, testing for conformance to a security policy, especially when this policy traverses several functionalities, and testing to reveal the presence of security vulnerabilities have not been widely explored in literature. This thesis focuses on these last two testing domains.

2.4.1 Privacy policies

Privacy policies are an example of the specification of high-level properties that traverse several functionalities. These high-level policies usually set the requirements for other more specific security policies, such as those governing authentication, access control and information flow [6]. There have been various works both in privacy modelling and in policies testing but these two areas have rarely been addressed together.

The first problem that has been studied is the formalisation of privacy policies. In addition to the classic languages for privacy policies [37, 102], other specialised access control languages such as XACML [132] have been used to describe privacy requirements [4]. It is interesting

that different languages, such as EPAL and XACML, share the same underlying structure and that their differences are mainly syntactic. It suggests that the research that has been carried out in the area of access control policies can also be adapted to be used in the area of privacy policies. Ultimately, one can achieve privacy by controlling who has access to which information in the system. The main problem with this approach is probably linked to the concept of *obligations*. This thesis discusses the research in testing access control policies and its application in testing general privacy policies later in this section.

Independently from specific languages, Barth et al. [11] present a framework based in contextual integrity to support policies enforcement. This framework is equipped with a modal logic for reasoning about privacy policies. This framework takes the basics of its model from Karjoth and Schunter [69] who present a general privacy model. Nevertheless, none of these two frameworks consider testing as an area of application.

Other formalisation efforts have been done with the aim of comparing privacy policies [9] or analysing obligations [65]. Others [82] have applied formalisms used to describe behavioural models, such as FSM, to model access control policies for, later, using known generation algorithms [33] to generate test cases that provide structural coverage for a defined policy. However, there is no previous work that formalises and uses a privacy policy to generate test cases for testing a system's implementation.

The general framework presented by Karjoth and Schunter [69] seems a good starting point to define a suitable way of describing privacy policies with the aim of generating test cases.

Extensive research has been done in the area of testing access control policies [82, 83, 4, 23, 24, 81, 78, 80, 79, 63, 105]. Most of them aim to evaluate access requests against policies (test case generation), while others compare versions of policies with each other and check policies for internal consistency [23, 24].

All the approaches cited above focus into testing a particular security mechanism - the *policy decision point* (PDP) - and verifying that the description of the policy itself represents what the requirements meant. For example, on one hand, Martin and Xie [78] present an approach for automatically generating test cases for a given access control policy. Their

approach considers each rule in isolation and attempts to satisfy the constraints required for that rule to be applied. Then, they measure the coverage of the generated test cases in terms of the number of exercised conditions. On the other hand, Martin et al. [81] use a random approach to generate requests against an access control policy. They test the correctness of the policy against its intended responses, not if an implementation meets the policy. They manually set some parameters to force the randomly selected test cases to provide condition coverage.

Those approaches rely on the assumption that there is a logical module - the *policy enforcement point (PEP)* - that interprets and enforces the execution of obligations. Paradoxically, there is no specific mention in literature about testing PEPs. The problem is that the PEPs can be implemented in different places in a system. Their functionalities can also be partitioned and split among different modules. Thus, even when policies are analysed and test cases are generated from them, there is no description of how these test cases can be linked to a particular system.

Only recently the problem of testing a PEP has been recognised [90, 91]. In their first work Mouelhi et al. [90] present a framework to qualify security test cases applied in the testing of the PDP and PEP modules of a system. They compare traditional functional test cases against test cases generated by an access control policy. The qualification criteria are based in the mutation testing approach. This work shows that in most cases there is a significant difference between functional and security test cases in terms of mutation scores. This work suggests that security specific test cases are needed and that they should be derived directly from the security policies, supporting the approach presented in this thesis.

In a second work, Mouelhi et al. [91] present an approach to transform functional test cases into access control test cases. This approach relies on the existence of a suite of functional test cases that cover 100% of the executable statements and proposes the selection of relevant test cases for which a new test oracle is generated. This new test oracle should verify that the security mechanism triggered by the policy works properly. This thesis addresses this problem for general privacy policies rather than specific access control policies. Moreover, the approach in this thesis does not rely on the previous existence of a suite of test cases. Rather,

it uses a behavioural model of the system to derive relevant test cases that exercise the rules and conditions in the privacy policy.

The concept of obligations is important in privacy policies [69, 61]. Privacy policies may not only grant access to data but may also make statements about actions that have to be performed. In simple terms, obligations are actions that are required to be performed before the execution of the test case is considered successful. Current approaches for testing access control policies disregard obligations claiming it is PEP's responsibility to enforce, and test, their execution. Only the approach of transforming functional cases into access control cases could address the testing of obligations. However, obligations are not mentioned explicitly in this work and the testing focuses in the interaction between the PEPs and the PDP only in terms of access permissions. This thesis considers that a complete test for conformance to privacy policies includes the testing of bounded obligations. Obligations, in general, are examples of liveness properties and this kind of properties can not be established by testing. Bounded obligations are those obligations whose execution can be verified in a limited period of time or after a finite sequence of actions.

2.4.2 Security vulnerabilities

Features and functionalities of every software program can be represented in abstract software models. In an ideal world the actual behaviour of the software is equivalent to the one specified by its model. However, in the real world software applications sometimes miss described functionalities and present some additional functionalities. A software *vulnerability* can be seen, in fact, as a system functionality that must not be present. This is illustrated in Figure 2.6.

Models are *abstractions* and as such they carry contextual *assumptions* about the environment. Consider an abstract model that specifies only the relation between inputs and outputs of a defined operation. This kind of model assumes that this operation will be used only to transform the specified set of inputs into their respective outputs. If the operation is used with any input that was not previously specified, then there is no way to know the correct output and any output could be acceptable. When a particular input is not considered in a

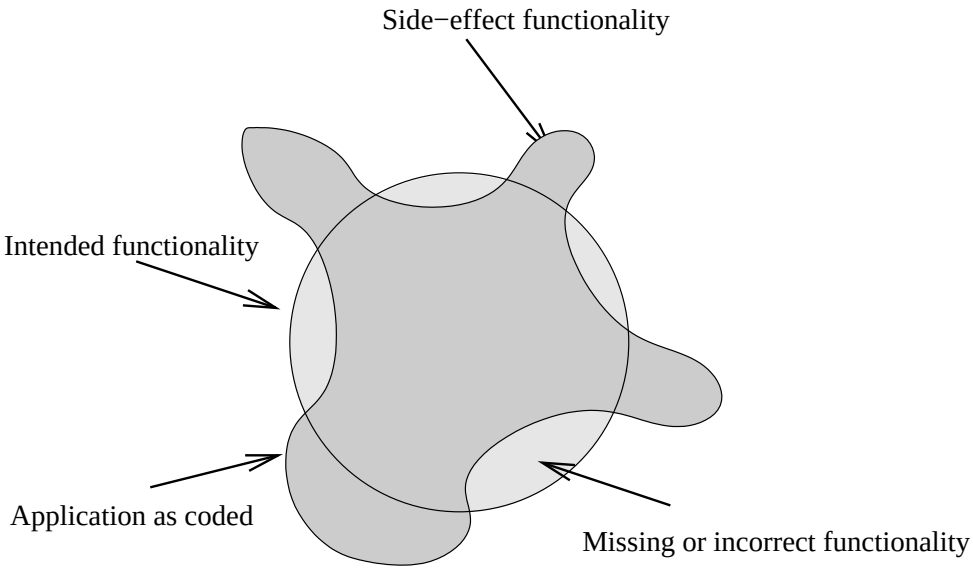


Figure 2.6: Intended vs. implemented software behaviour (source: [115]).

model, this model is *underspecified*. Underspecification is inevitably linked to abstraction.

Most security problems are due to underspecified functionalities triggered by underspecified inputs. In general, under-specified functionalities are added by programmers during or at the end of the coding phase. Moreover, usually, functional testing of parts of the system related to these inputs is not conducted. This is where vulnerability testing is useful, testing the product for harmful additional behaviour.

Before continuing on discussing vulnerability testing, consider the work of Fithen et al. [44] with the aim of formalising the definition of *vulnerabilities*. They define vulnerabilities as unplanned system features that an intruder may exploit, if certain preconditions are established, to achieve particular impacts on the system that violate its security policy.

In the previous definition, preconditions refer to the set of conditions that must exist for an event to occur, the exploitation of the vulnerability in this case, and impacts refer to the conditions that will exist as a result of the vulnerability exploitation. Seacord and Householder [109] take this definition and separate explicitly the definition of vulnerabilities from the definition of an *exploit* (the exploitation of a vulnerability). In the same way, this thesis defines a *vulnerability* as being the set of preconditions that enable an exploit and an *attack* as the technique that takes advantage of a vulnerability to violate an explicit or

implicit security policy.

Although the literature agrees on the definitions, there still seems to be little or no consensus on the way security properties and vulnerabilities have to be modelled. However, recent works ([32],[133],[12]) tend towards formal methods for modelling security properties. According to Chinchani et al. [32], formal models offer an elegant solution to vulnerability assessment. Although agreeing with the previous assertion, this thesis still points out that among formal models, different notations or languages have been used and developed. Maybe the difference resides on what (which security goal, a security functionality, an attack or vulnerability) is going to be modelled.

Using the formalisms and notation described in Section 2.1 this thesis defines vulnerabilities and their associated concepts as follows.

Definition 7 (Vulnerability) *A vulnerability is a state predicate that defines the conditions that enable an attack.*

Definition 8 (Vulnerable state) *Given a set of states Q in a model SE that includes details of the SUT and its environment, and a state predicate v that represents a particular vulnerability, a state $q \in Q$ is vulnerable if $v(q)$.*

Definition 9 (Attack) *Given a vulnerability v and a state predicate i that represents the (malicious) desired impact on the system, an attack is a path from q to q' where $v(q)$ and $i(q')$.*

With the previous definitions this thesis defines its focus on vulnerabilities that arise from defects in the software itself and its configuration, that is, how the software is "connected" to its environment. These definitions also provide the basis for including these concepts into behavioural models and providing solutions to some of the problems in current security vulnerabilities testing.

Traditionally, security vulnerability testing has been done using penetration testing [8]. In this approach, once a software product is finalised, a group of security consultants subject the software to a set of (malicious) attacks and verify the software's resilience to such attacks.

One of the main problems of penetration testing is that its success depends on many factors, few of which can be measured and standardised. The most obvious variables are tester skill, knowledge, and experience. Without a standardised procedure and without adequate documentation of security requirements, assumptions, threats and attack patterns, security findings can't be repeated across different teams and vary widely depending on the tester. This thesis explores the assumption that models can provide an adequate way of documenting security properties, assumptions, vulnerabilities and attacks.

In the way this thesis sees it, a security model is a framework for understanding and solving the problem of security for a particular purpose. A large list of software vulnerabilities has been compiled and classified through the years with the aim of avoiding these vulnerabilities in future developments [74, 97, 88, 34]. The purpose of vulnerability models is to serve as a tool for verifying the absence of these vulnerabilities in software applications.

State transition based approaches have been used before to model other security related properties and applications. Examples of these are security protocols and penetration attacks. Many network security related properties, such as confidentiality and authentication, can be verified by analysing the traces of a state transition system and determining whether the properties are preserved for each trace or not.

Attack graphs [100] represent a popular state-transition model for security problems analysis. In this approach, all possible actions are represented as transitions while different states of a network are defined by the edges. The goal is to find a path from an initial state (considered safe) to one or more states where security is compromised. These paths are considered possible attacks. The work by Sheyner et al. [110] is the first formal treatment of attack graphs. They showed that model checking can be used to automatically generate attack graphs. Their modified model checker was able to output all the counter examples (all the possible attacks), thus, enabling the tool to show all the multi-stage, multi-host attack graphs. However, this approach suffers from state explosion and this affects its scalability.

Petri nets have also been used for security modelling. A Petri net consists of places, transitions and tokens. Places are represented by circles and can be compared to nodes in a graph. Action or flow is modelled with tokens that move between places along transitions.

Transitions are represented by rectangles and connected to places by arcs. Transitions have AND semantics, meaning that a token is passed only if a token is available at each of the input arcs. McDermott [84] used Petri nets for modelling attacks in an approach he called *Attack Net*. In this model, attack steps are represented by places similar to nodes in attack trees. Transitions are used for the explicit modelling of attacker actions which extends the expressibility of this model compared to attack trees. Helmer et al. [55] extend the expressibility of attack modelling using Coloured Petri Nets. A CPN is a Petri net in which tokens are associated with tuples ("colours"). These tuples contain additional information for the objects of an attack. In this way, for example, when a transition requires the input tokens to appear in a defined order, they can carry with themselves a time mark. This mark allows the transition to verify the order of appearance. Recently, Xu and Nygard [133] have used Petri nets to model software applications, threats and threat mitigations (security countermeasures) as a whole. They extended the classic transition-based modelling of Petri nets with concepts from the aspect-oriented paradigm. Their approach uses three models and is centred on verification. An important condition of this approach is that the three models must be in the same level of abstraction in order to be combined.

Several authors [133, 129, 114] agree on the use of the term *threat* as a condition that precedes a vulnerability. Security threats are potential attacks, i.e., misuses and anomalies that violate security policies. Security threats are always present, and vulnerabilities only appear when the threats have not been mitigated by security countermeasures (security functionalities or assurance techniques). A useful technique related with the concept of threats is *threat modelling*. Threat modelling takes a data flow approach to application security. It helps the testers understand where input comes from and how an adversary can manipulate it to cause the system to fail. Moreover, a threat model describes a number of actions that an attacker may take to exploit a vulnerability of the system. This thesis explores the composition of a threat model with the behavioural model to verify if a given implementation enables (inadvertently) a specified attack.

Modelling approaches differ in the kind of information the models contain. As a summary one can argue that the relevant information to be recorded in an attack (or vulnerability)

model for security purposes consists of the conditions under which an attack can be performed, the capabilities and resources that are required, the steps of the attack (and their order in time), the interdependencies of the attack steps, and the effects on the SUT after the attack has been performed.

2.4.2.1 Other related domains

Above it has been described that modelling efforts cited in the literature have concentrated on specifying security policies, especially access control policies. However, much effort has also been put into the specification of protocols. In a daily increasing networked environment it is necessary to secure data travelling among the network nodes and also to certify that the network components communicate in a secure manner. Driven by the need for communication, research in protocol modelling and testing has proceeded along a separated and privileged trail with respect to software testing [15]. The following are provided as examples of these research efforts.

- Security requirements of protocols can normally be expressed as correspondence assertions or as constraints on protocol traces. These assertions may also contain references to the past or the future. Thus security protocols have been specified using logic languages, such as the language described in [35], and languages based on process algebras like CSP or CCS, such as the one used in [106]. Some specification mechanisms have explicitly used temporal logic for reasoning about the past and future and others have captured this in the semantics of the logical predicates.

- Other specification languages that deal with the state of the system have also been used to model security protocols. The Z language, for example, was used by Long et al. [77] for modelling cryptographic protocols involving most of the common cryptographic operations such as symmetric and asymmetric encryption, message digests, and nonces. The central idea of Long et al.'s work is that security requirements can be specified by Z-invariants, using disjoint types of data for specifying keys, nonces, and encrypted data, mathematical functions to represent cryptographic operations, and logical constraints

over the data and functions.

It seems worthwhile looking at the advances and results in protocol testing. Although these results have been conceived for a very specialised area, several ideas can be adopted and adapted for security testing, particularly to exploit the advances in model-based testing.

2.5 Concurrent asynchronous systems

Concurrent systems are those systems in which two or more components, threads or distributed components, for example, are active simultaneously and interact with each other [36]. One way in which components in a concurrent system interact with each other is by exchanging messages. Asynchronous systems are concurrent systems that read and respond messages as schedules permit. There is no assumption of a global clock or ordering of execution among any two components (the sender and the receiver) [56]. This means that a component may send a message at any time, and may receive a message at any time, and also that a message that arrives to a component would have to wait an undefined period of time to be processed. In the meantime, as it will be not efficient to wait for an answer the sender of the message will continue performing other tasks.

There are many research works that aim to describe asynchronous systems to obtain a greater understanding of their properties and/or perform their testing. Each of these works use different techniques to represent the systems. For example, Tretmans, in one of the most cited works on the area [118], uses *labelled transition systems* to describe asynchronous systems, Campbell et al. [28] uses *interface automata* and Maréchal et al. use *symbolic transition systems*. These last two formalisms are based on LTS. Others that use LTS are Veanes [124], that extend the concepts of refinement and alternating simulation for their use with asynchronous systems, and Bhateja et al. [16] that introduce different equivalence relations.

Bhateja et al. [16] review and analyse some recent work in the area of testing for asynchronous system. They work on the settings of labelled transition systems, inspired by the earlier work of Tretmans [118]. To differentiate between actions executed by the SUT and

actions executed by some element of the environment, they define input and output actions. They use queue semantics to "postpone" input actions and define the equivalence of two systems if one can be transformed into the other by means of postponing input actions. The use of a queue semantics limits the applicability of this approach to channels in which only communication delays occur. That is, it allows different interleavings between input and output actions but requires the order of each subset of actions to be maintained. This thesis, in contrast, proposes the use of a set semantics in order not only to postpone observable actions but also to deal with imperfect channels where the order of observable messages is not guaranteed. Moreover, the practical approach in this thesis complements the theoretical approach of Bhateja et al. [16] by defining algorithms to perform the generation and execution of test cases.

Along with transition systems, Winskel [131] describes other models used for describing asynchronous systems, such as synchronisation trees [87] and Hoare traces [21]. These models are called *interleaving models* because they identify concurrence or parallelism with nondeterministic interleaving of atomic actions. They abstract away the fact that the systems are composed by several independent computing agents and model the behaviour of the entire system in terms of purely sequential patterns of actions.

Winskell [131] also presents a different kind of models, called *non-interleaving models*, such as *Petri-nets* [95] and *event structures* [130, 123]. These models are not as abstract as the interleaving models and maintain information on the (physical) separation among the components of the system. Nevertheless, Winskell himself shows relationships among the different models and how interleaving models can represent the same properties as non-interleaving models by integrating elements such as a naming system for actions. Others have also worked on the transformation of one formalism into another. Here this thesis presents some examples with emphasis on event structures. Henigger [57] generates an equivalent prime event structure from a system of asynchronously communicating state machines. Nakata et al. [94] link context-free processes specifications to the concept of symbolic event structures. They use the properties of event structures to derive the protocol specification of a distributed system. Herbreteau et al. [58] present an algorithm to generate labelled event structures from

a well-structured LTS specification. In principle, the well-structured condition is necessary to deal with infinite state systems.

Not specially linked to any formalism, Hendrickson et al. [56] present a technique to build the description of the system from the observation of the message exchange occurring among the components of the system. The interesting part of this work, from the perspective of this thesis, is the definition of the observed elements; events, representing the execution of actions, and the relationships among these events, especially the causality relationship.

With respect to testing and validation, asynchronous systems present challenging characteristics, such as local non-determinism and communication delays. These characteristics have been addressed theoretically in various research works [118, 31, 18]. Tretmans [118] introduces a so called "queue semantics" where two output traces are considered equivalent if one can be transformed into the other by postponing the occurrence of output actions. No reordering among output actions is permitted. This idea exploits the concept of *quiescence* in LTS. Boreale [18] et al. propose the use of a bag instead of a queue for the case of parallel output actions. Effectively they allow the reordering of actions that are executed in parallel. This occurs mainly to reduce the number of interleavings in the model and to allow different components to be modelled independently. Nevertheless, both research works assume the use of perfect communication channels, thus output actions modelled in a defined sequence will always be observed in that sequence. However, from a practitioner's point of view, this assumption does not always hold. The modelling and testing of asynchronous systems in this thesis does not assume perfect communication channels. Here, the idea of Boreale et al. for parallel actions is extended to sequential actions under the assumption that a component can observe actions in a different order from which they were executed, due to delays and losses in the communication channels.

Several testing tools, among which we can cite TGV [67] and TorX [14], have been used in experiments with distributed and synchronous systems. TorX uses an on-the-fly testing approach. In [14] TorX is used to test a *Conference Protocol*. Contrarily to the assumptions in the present thesis, the experiment in [14] assumes that the communication is reliable and message exchange can be modelled as (FIFO) queues. TGV, on the other hand, requires

test cases to be serialised first, consequently, it cannot deal directly with concurrence and introduces unnecessary synchronisations between distributed testers.

Finally, the work in this thesis is based on very well defined conformance testing theories and conformance relations such as IOCO [117]. However, the systems we are interested in are not always *input enabled*. That is, some inputs are only enabled by causality relations with other inputs or outputs. Implicitly this thesis implements the idea of Boreale et. al [18] that process input cannot be forced. Formalisms such as *IOLTS*[117], *trace automata* and *concurrent transition systems* [111] have also been used to model concurrent systems. All these formalisms have a basis into LTS. Therefore, the use of LTS in this work is general enough to support other formalisms and integration with current approaches.

2.6 Testing tools

Given that the main objective of this thesis is to define the characteristics of a framework that supports the automation of the testing process, hereafter we present a set of tools for test generation and/or execution. The list of tools mentioned here does not pretend to be exhaustive and the only criterion for its selection is its availability.

2.6.1 TGV/CADP

TGV [67] stands for Test Generation with Verification technology. It is tool conceived to generate test suites for protocols. TGV uses models described in several specification languages. Those languages include Lotos[17], SDL, UML and IF[19].

TGV uses test purposes not only to determine the coverage criteria but mainly to direct the generation towards "interesting" functionalities. With this objective in sight, TGV takes as input two Input Output Labelled Transition Systems. One IOLTS describes the specification while the other formalises the behavioural part of a test purpose. The tool computes the synchronous product of both IOLTS following the approach in [30]. This product results in a new IOLTS where some states are labelled as accepting or rejecting states based on the information contained in the test purpose. Test cases are then generated by selecting accepted

behaviours.

TGV offers the possibility of generating a complete test graph that contains all test cases corresponding to the test purpose, or generating individual test cases. In the complete test graph, *pass* verdicts are based on traces that reach accepting states. Traces that do not lead to an accepting state are truncated with an *inconclusive* verdict. Any other trace that is present in the implementation but not in the test graph receives the *fail* verdict.

This tool produces test cases described in an early version of the TTCN standard [47]. To achieve this, a depth-first searching algorithm transforms the IOLTS into a test graph that represents the observable behaviour of the system. Message parameters are obtained form the labels in the original IOLTS.

2.6.2 TorX/CADP

This tool is representative of the family of test generation tools based on an Input- Output Labelled Transition System (LTS) model of the system under test. It accepts models written in the Promela and Lotos languages. It was designed for conformance testing of reactive systems.

TorX [14] requires a real implementation and a formal specification of that implementation. It works as the arbiter that checks the correct behaviour of a real implementation during its execution based on the formal specification. It combines test generation and test execution in an integrated manner. It uses the *on-the-fly* testing approach, i.e. instead of generating the entire test case and then applying it, the tool generates only the next event that is available according to the specification and immediately executes it.

The TorX test generation algorithm is based on a walk through the state space of the specification. This walk can be done randomly or controlled by the test purpose, which is anything that represents a set of traces over the model. A test purpose acts as a test case specification and makes it possible to drive the random walk.

2.6.3 Microsoft Spec Explorer

Spec Explorer [28] is a model-based testing tool initially developed at Microsoft Research as a result of continuous research in the area [10, 49, 92]. Spec Explorer enables modelling and automatic testing of diverse kinds of systems, including concurrent object-oriented systems. We distinguish between two versions of this tool. The first version [28] developed as part of a research project and not available for commercial uses, and the new implementation, offered to be released in conjunction with Microsoft's development suite – Visual Studio 2010. Hereafter we refer to the original, or first, implementation unless the contrary is explicitly stated.

Spec Explorer used *model programs* to express the system behaviour as *abstract state machines* [48]. A model program is a program, much smaller and simpler than the real implementation, that usually defines a subset of the implementation's features. The original implementation of Spec Explorer used a specific language, Spec# (an extension to the language C#) to write the program models. The new implementation will use program models written in C#. The code written in C# will be complemented by a *coordination language* (known as "Cord") which will provide features to combine models, generate test data and select certain scenarios that are especially relevant for testing.

A model written in Spec# contained the definition of classes and variable types. Instances of these classes and types defined the state of the system. The model also contained actions. Actions were methods with preconditions that defined in which state of the system they may occur and for which input parameters.

The model program defined, by exploration, an interface automaton over which an explicit state model checking algorithm was used to compute the (possibly infinite) space of all possible sequences of method invocations. Method invocations were restricted to those that do not violated the pre- and post-conditions nor the invariants of the systems. Invariants were properties of the system defined to hold always, before and after the execution of an action.

To avoid, or at least soften, the problem of state-explosion, Spec Explorer included two mechanisms to reduce the size of the automaton. First, it provided the tester with the possibility of defining an equivalence relation over states. In this way, Spec Explorer performed *state grouping*, pruning away states that even when distinct were undistinguishable from the

tester's perspective. Second, it implemented *state-dependent parameter generation* which allowed it to compute the parameter domains of each action with respect to the current state. This eliminated the possibility of generating infeasible states.

In Spec Explorer, test cases were generated by traversing the graph of the interface automaton. Test cases were evaluated under the assumption that the SUT must accept at least as many inputs as the interface automaton defines and that, conversely, the automaton must accept at least as many outputs as the system may produce. More formally, there was a relationship of *alternating refinement* between the model and the SUT.

Finally, with respect to the execution of the test cases, Spec Explorer offered two approaches, on-line and off-line test execution. In off-line test execution, a standalone test suite was generated with complete behavioural coverage over a restricted domain of system inputs. In on-line testing, Spec Explorer exploration generated the next actions and their parameters based on the observed history of the test run. This last feature is very useful for testing non-deterministic systems.

2.6.4 SmartMBT

SmartMBT* is a model-based testing tool implemented in Prolog. Its host language provides the tool with necessary features like backtracking. Its test case generation process is based on the theory of testing with labelled transition systems. It implements widely used algorithms and approaches. Its test generation, for example, measures transition coverage, and uses the Chinese Postman algorithm to produce test sequences that cover all the transitions in the LTS.

SmartMBT takes as input an LTS represented by the set of state variables and the set of actions of the system. The states of the LTS are ultimately the product of the possible values of the state variables. The transitions of the LTS are represented by Prolog predicates that encode pre- and post-conditions for a defined action. Pre-conditions are interpreted as guards rather than as actual pre-conditions in the sense they are described in the process algebras [62]. That is, if the model gets into a state where the pre-conditions of a given transition do

*developed by KJRoss and Associates

not hold, then the action represented by this transition will not be available for execution. Contrarily, in standard process algebras, if pre-conditions do not hold, then the action can still be executed, but no guarantees are given with respect to the post-conditions (state of the system after execution).

SmartMBT implements three algorithms for generating test sequences. The first one completely automates the process by using the Chinese Postman algorithm. It uses the LTS and a transition coverage criterion to drive the generation of sequences. If a reset operation is available on the model, one sequence will be generated so that it covers all transitions of the underlying LTS. If no reset operation is available, several sequences will be generated, the change between one sequence and another working as a reset.

The second algorithm is an implementation of *random walks*. Traditional random walks techniques such as the *transition tour* (TT) method [93] are driven by coverage. In SmartMBT the random generation algorithm is not driven by coverage but driven by the tester. The tester can limit the number of steps in a sequence as well as define *weights* for each action. The weight of an action defines a relative probability of execution, i.e., actions with higher weights will have higher probabilities of being executed than those with lower weights.

The third technique implements an interactive approach. In this technique, sequence generation is driven by the user one step at a time. That is, given a defined state of the model, the tool calculates the set of executable actions and presents it for the tester to choose which action to execute. Once the tester has made a choice the tool calculates the next state of the model and starts the process again. The random and interactive approaches can be interleaved.

SmartMBT also implements both approaches for test case generation, off-line and on-line generation. For the off-line approach, sequences are generated uniquely from the model taken as input. The on-line approach requires the implementation under test (IUT) to be executed together with the SmartMBT tool. It also requires the tester to provide an extra module to transform action labels from the model into executable calls to the IUT. When this approach is used, for each action executed on the model, SmartMBT checks that the same action can be executed on the IUT. A verdict, pass or fail, is produced for each step of the sequence.

Chapter 3

Test automation framework

This chapter presents the concepts and elements that are required in an automated process of testing software. It includes a description of these elements and the relationships between them in terms of a simple and general structure that constitutes a framework for test automation. The focus of this chapter is on the concrete way in which these elements can be linked together to enable the (model-based) testing of a broad spectrum of software systems.

3.1 General framework

In a model-based testing framework there are two central elements: the requirements specification and the implementation. Requirements are specified by models and the implementation is known as the SUT. The present testing framework uses the term models in a general way to refer to specifications. It avoids the usual dichotomy of model-based and property-based specifications [107]. Moreover, complex systems and properties require the use of different specification paradigms to be described completely. The present framework allows the use of different kinds of specifications in a unified fashion.

In the present testing framework, a model specifies a set of behaviours that a system can exhibit. A behaviour at the model level is described as a sequence of actions where the order in the sequence implicitly defines the rules and conditions for actions to be performed. Thus, models specify what actions a system can perform and when or under which conditions they

can be performed.

In an abstract way, actions refer to the central elements of most specification paradigms. For example, they represent events in a history-based specification, identify uniquely a transition function in a transition-based specification, represent processes in operational specifications, or define the time of a particular snapshot of the system in a state-based specification. Therefore, the present testing framework allows the tester to choose among different modelling languages or notations in order to describe the models provided that there exists a definition of how these models map to sequences of actions.

Ideally a single model specifies all the behaviours of an entire system. However, as systems turn complex, any attempt of representing all, or most of, their features in a single model will produce a complex model. In order to manage complexity, different modelling languages and notations tend to specialise on some characteristics of the software while disregarding others. Moreover, even when specified using the same language or notation, models are sometimes split to represent different parts of the system. Thus, a software system is specified using a collection of different models which in turn can be written in different languages and can define the system at different levels of detail.

In a general way, this testing framework addresses the separation of the control or operational part of a system from the data the system uses or processes. Models that represent the control flow of the system are called *behavioural models*. Behavioural models disregard details that refer to data handling to make themselves more manageable. All data related specifications are handled by another group of models called *data generation models*.

This testing framework uses behavioural models to derive abstract test case specifications from them. These abstract test cases represent an abstract behaviour of the system (or a set of them) and are a formalisation of the objectives defined for the testing process. Abstract behaviours represent a defined flow of control in the system but still disregard most data specifications.

The testing framework composes abstract behaviours with the data generation models to specify (concrete) behaviours. Concrete behaviours define the control and data flow of the system. Thus, the composition of these models leads to the generation of behaviours

with values. Behaviours with values are not only more specific representations of how the system behaves, but also solve non-determinism that arises from the use of abstract models. Behaviours with values lead to the generation of test cases, which is the first objective of a model-based testing framework.

The process of generating test cases has a core element which is the *generation algorithm*. The generation algorithm is usually tailored to suit different testing objectives and types of models. This process not only derives a valued behaviour from the system's models but additionally defines the *oracle* of the test case. In an abstract way, the oracle is a predicate over the system's behaviour which is evaluated after the execution of the test case.

The second objective of a model-based framework is to execute generated test cases to produce a verdict. This thesis presents a testing framework that is capable of executing concrete test cases in an automated fashion. Therefore, the testing framework defines a link between the generated test cases and the SUT. For a specified software system, the models and the SUT are linked by a relationship described as a mapping. Figure 3.1 shows a pictorial representation of this mapping relationship. This relationship links *actions* at the model level to *operations* at the implementation level.

At the implementation level, operations represent functions or procedures of common programming languages, calls to APIs, or available functionalities inside a GUI. At this level, a behaviour is represented as a sequence of operations. To represent behaviours as sequences seems natural as the latter are present on all specification formalisms cited before. Additionally, it enables the framework to define naturally the fact that a single action at the model level can map to a sequence of operations at the implementation level.

In the following sections, this chapter presents a more detailed description of the core elements of the framework, namely the models, the test generation process and the test execution process. Then, this chapter also presents a tailored description of these elements for different *testing domains*. A testing domain is defined by the particular characteristics of the SUT and the defined objectives of the testing process.

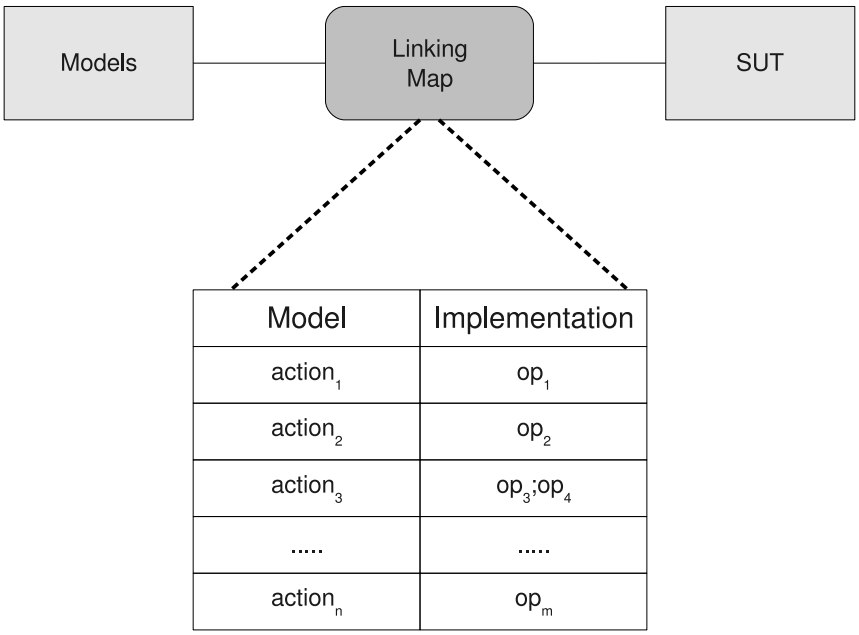


Figure 3.1: Relationship between models and SUT

3.2 Models

3.2.1 Behavioural models

In an abstract way, a behaviour is defined as the change of state in a system that is triggered by the execution of an action; therefore, the central elements to build a model that describes behaviours are states, transitions and actions. Subsequently, the underlying formalism behind the models in this framework is based on *labelled transition systems* (LTS).

Hereafter a model M refers to a LTS with elements $(Q, A, \rightarrow, q_0, F)$ where Q is the set of states of the system, A is a set of action labels, $\rightarrow$ is the transition relation $\rightarrow \subseteq Q \times A \times Q$, q_0 represents the initial state of the system and $F \subseteq Q$ is the subset of accepting states of the system. Note that the subset of states F is an addition to the standard definition of LTS shown in Section 2.1.1. This subset implements the concept of *quiescent states* defined in Section 2.1 and, in general, represents the knowledge of the tester of what is considered an acceptable behaviour of the system and what is not.

In general, a model represents a set of behaviours. The model of a software system represents all the possible behaviours of that particular system. For test automation, it is desirable to be able to describe a particular behaviour, which at the model level is called an *abstract behaviour*. An abstract behaviour is just a path in the (abstract) model, however for pragmatic reasons this thesis separates the trace of the path from the sequence of states, resulting in the following definition.

Definition 10 (Abstract behaviour) *An* abstract behaviour *in the model M is a pair of sequences (σ, α) where*

- *σ is a sequence over Q, and*
- *α is a sequence over A;*

such that for every j in $\{1 \dots n\}$, where n is the number of actions in α, $q_{j-1} \xrightarrow{a_j} q_j$.

In an abstract model, and extensively in an abstract behaviour, an action a is said to be *enabled* if given the current state of the system q, there exists a state q' such that $q \xrightarrow{a} q'$.

At an abstract level, the present testing framework does not define what a state is or how it is represented. This gives flexibility to the tester at the time of selecting a suitable notation for the models. However, this framework assumes that during the test generation process, implementation details are introduced into the models to make them more specific. These implementation details are represented into the states of the system as a mapping between a set of variables SV, also known as the state variables, and a set of values Val.

The mapping between state variables and values could be specified into the behavioural models themselves. However, behavioural models usually elide specification of values to maintain simplicity. Therefore, the present framework defines a data generation model that is described separately. This separation between behaviour and data makes model-based testing tractable. Now, this thesis presents a detailed description of these data generation models.

3.2.2 Data generation models

The present framework uses context free grammars with rules guarded by predicates as a mechanism of generating test data. The guards enable or disable a specific production rule. This allows different elements of the framework and the tester to direct the generation of test data by modifying the context in which these guards are evaluated. Additionally, this data model allows TLA [73] like actions to be associated with each production rule. The actions can modify the context associated with the data generation. This allows the generation to be influenced by the history of previously executed actions and generated data. The elements of the data generation models are defined more precisely as follows.

First of all, the values to be associated with an action not only depend on the internal state of the system but mainly on the state of the system's environment. Thus, the framework assumes a set of global variables V that includes the set of state variables of the system SV as well as new variables necessary for data generation. A particular assignment of values to these variables is called a *valuation*. Towards controlling the data generation process the framework defines a *control statement* to be a predicate/action pair over V. This is written as $\{g; \xi\}$ where g is a predicate and ξ is an action (as per the TLA definition) over V. The framework assumes that the action cannot modify any variable that is part of the models

of the system either at the behavioural or implementation levels. This is to ensure that the integrity of the model is not violated by the data generation process. Technically this can be written in TLA as "unchanged SV". However, it is not explicitly written inside the specifications but implicitly considered in any data generation model. The symbol ρ denotes a particular assignment of values to the variables in V.

Next, this thesis defines the extended context free grammars and how they derive values.

Definition 11 *A guarded grammar G is a tuple (N, T, S, P) where N is the set of non-terminal symbols, T the set of terminal symbols, S belonging to N the initial symbol, and P is the production relation. Each element of P is a member of*

$$BP \times N \times (N \cup T)^*$$

where BP is a control statement.

Typical elements are written as $\{g; \xi\}\gamma \to \delta$ where $\gamma \in N$ and $\delta \in (N \cup T)^*$

A one step application of a production rule and its generalisation to multi step applications in the context of a valuation is defined below.

Definition 12 *We write $\delta_1\gamma\delta_2 \ \to_\rho (\delta', \rho')$ if*

$\{g; \xi\}\gamma \to \delta$ is a production rule such that g is true in ρ and

$\delta' = \delta_1\delta\delta_2$ and ρ' is the result of executing ξ in ρ.

A multi step application $\delta \Rightarrow_\rho (\delta', \rho')$ is performed if

there are sequences $\delta_0, \delta_1 \cdots \delta_n$ and $\rho_0, \rho_1 \cdots \rho_n$ such that

$\delta_0 = \delta$, $\delta_n = \delta'$, $\rho_0 = \rho$, $\rho_n = \rho'$ and

for all i between 0 and $n - 1$ $\delta_i \to_{\rho_i} (\delta_{i+1}, \rho_{i+1})$

Using these extended grammars the present testing framework derives the values needed to make behavioural models and abstract behaviours more specific or concrete. In the following sections, this thesis defines how these grammars are composed with the behavioural models.

3.2.3 Composing behavioural and data models

Given a behavioural model $M = (Q, A, \rightarrow, q_0, F)$, the testing framework associates a grammar with each action in A. In this way, the framework uses the generated values to make each abstract behaviour at the model level more specific. In the framework there is not a unique grammar G_a but a collection of grammars for each action a in A. Each grammar G_a defines a different set of production rules so that the same behavioural model is used for generating tests with different data values.

The association between behavioural models and grammars leads to the definition of a global model. In this global model the framework defines a global state that represents an abstract behaviour of the system jointly with a sequence of values and the current valuation of the set of variables V. A global state represents a (concrete) behaviour of the system. The global model modifies its state by extending the current behaviour of the system. The following definitions formalise these concepts.

Definition 13 (Global state) *A global state is represented by a tuple written as* $(\sigma, \alpha, \omega, \rho)$ *where*

- σ *is a sequence of model states,*
- α *a sequence of model actions,*
- ω *a sequence of values, and*
- ρ *the current valuation of the set of variables* V*.*

Definition 14 *The initial global state is* $(q_0, \epsilon, \epsilon, \rho_0)$ *where* ρ_0 *is an initialisation of the variables in* V*.*

The extension of a given global state defined as a transition is presented below.

$(\sigma q, w, \alpha, \rho) \rightarrow (\sigma q q', wa, \alpha v, \rho')$ *where*

$q \xrightarrow{a} q'$ *and*

$S_a \Rightarrow^*_\rho (v, \rho')$ *where* $v \in T_a$

The above definition states that the initial global state has no generated action or symbol and the model is in its initial state with the global variables having a suitable initialisation. Whenever the action a can be exhibited by the behavioural model, the grammar G_a is required to generate a value (v) to be associated with the action. The grammar G_a is always run from the initial symbol. The valuation of the global variables are used to guide the derivation of the value v.

From the composition of behavioural and data generation models, behaviours with values can be derived. These behaviours are the core components of test cases. This thesis will now describe the process of selecting one or more behaviours to compose or generate test cases.

In general, an action a that can be exhibited by the behavioural model is said to be *enabled*. For practical purposes, however, an action a is enabled only if the global state can be extended using a. A function *enabled* over a behaviour and a single action that indicates whether this action is enabled by the given behaviour, is formally defined as follows.

Definition 15 (Enabled) *Given a model M, a behaviour $\mathcal{B}_M$ represented by the global state $(\sigma q, \alpha, \omega, \rho)$ and a single action a in A,* enabled$(\mathcal{B}, a)$ *holds* ***iff*** *there exists v such that $(\sigma q, \alpha, \omega, \rho) \rightarrow (\sigma q q', \alpha a, \omega v, \rho')$.*

In addition to the previous definition, consider also the function *lastState*$(\mathcal{B})$ that returns the last state in the sequence of states σ of the behaviour $\mathcal{B}$. Hereafter when this thesis refers to a behaviour it refers to the respective global state.

With the previous definitions, this thesis now defines the process of generating test cases by extending the initial behaviour of a system.

3.3 Test generation

Test generation is the process that uses a set of models to produce a set of behaviours representative of a defined software property. These behaviours are then executed against the SUT and the result of their execution is validated to verify the defined software property. The set of generated behaviours is called a *suite of test cases*. A test case is, then, linked to

a (concrete) behaviour and provides a mechanism for validation. The testing literature refers to this validation mechanism as the *test oracle*.

The present testing framework provides, in general, a way of representing behaviours in terms of labelled transition systems. The framework requires also a suitable way of describing the test oracle. The testing framework uses the concept of *state predicates* (defined in Chapter 2) to represent this oracle. The set of constants and function symbols, and their semantics, needed to describe the state predicate depend directly on the system under test and the testing domain. Considering the above described elements, the testing framework defines a test case as follows.

Definition 16 (Test case) *Given a model M that represents the possible behaviours of a software system, a* test case *is a pair $TC : (\mathcal{B}_M, sp)$ where $\mathcal{B}_M$ is a behaviour of M such that the last element in the sequence of states is q_j and sp (the oracle) is a state predicate such that $sp(q_j)$ holds.*

Given a model based on a LTS, the goal of a test case when executed is to show the presence (or absence) of a defined transition in the SUT. Thus, in abstract terms, the test generation process

- identifies the transition(s) that represent the testing objectives,
- searches the models for this transition, and
- produces test cases that include the behaviours that lead to the desired transition(s).

Testing objectives are a set of required properties of the SUT. On one hand, these properties can be implicitly included in the algorithm that represents the test generation process. For example, consider a coverage criteria that requires all transitions of the behavioural model to be tested, the test generation will include some "touring" algorithm such as the Chinese Postman algorithm. This algorithm will select a particular behaviour that contains all the required traces. On the other hand, the properties of interest can be included in a model that when composed with the behavioural and data generation models show the required transitions for the test cases. This approach implements the concept of *testing purposes* [38].

Independently of which approach is implemented to generate test cases, behaviours inside the test cases are executed against the SUT by the test execution process. This thesis will now discuss the generalities of this process.

3.4 Test execution

In general, test execution refers to the process of taking a (concrete) behaviour and executing its sequence of actions with their correspondent values as parameters using the proper SUT interface and operations. Additionally, it is expected that the results of an execution will be compared with the expected results. The expected results for each action execution are defined by the sequence of states on the behaviour. The overall expected result of a test case is defined by the state predicate (oracle) of the test case.

The test execution process for the present testing framework is composed of four different steps or phases:

- The first step is the implementation of the map between actions and operations and the implementation of the Adaptor scripts that communicate directly to the SUT's interface. This step follows the action-word approach described in Chapter 2.
- The second step is the actual execution. This execution uses the previously implemented scripts to control the SUT and is performed following an on-line or an off-line approach. Both execution approaches are described in Chapter 2.
- The third step is validation. Generally, the Adaptor scripts contain *assertion* clauses that determine whether the oracle predicate holds for the current state of the SUT or not.
- Finally, there is the update of the state of the model and the global (environment) state.

3.5 Concluding remarks

This chapter has described a general testing framework based in models. It has focussed on the separation between the behavioural models and the data generation models. This separation

enhances mainteinability and reusability of the models, mainly of the behavioural models. Behavioural models have been described using LTS. This description allows the modeller to concentrate on the high-level functionalities (actions) of the SUT and how and when they become enabled in order to be used. Specifically this is how and when the user can access them by using the different interfaces of the SUT. Data generation models are described using extended CFGs. Different from traditional CFGs, the extension used in this thesis provides each production rule in the grammar with a logical predicate, the *guard condition*, and actions, as per TLA definition. The actions allow the update of the current global state of the system. Actions in the data generation model are not allowed to modify the state of the behavioural model, thus the integrity of the behavioural model is not compromised.

This chapter has also defined the way in which behavioural and data generation models are composed so that they can be used in the test generation and execution processes. The composition of these models gives place to an implicitly defined global model of the system. In this global model, each state is defined mainly as a behaviour (in the behavioural model) with values (provided by the data generation model). The valuation of global variables in a global state represents properties of the behaviour contained in that state.

Finally, the process of generating a test case is defined, in simple terms, as the process of "walking" over the global model, effectively, extending a given initial behaviour towards the desired behaviour for the test case. The way in which this walk is performed is dependent on the type of system and the testing objectives. In the following chapter this thesis will discuss how this process is specialised for different types of systems and testing objectives.

Chapter 4

Framework specialisation for different testing domains

The processes of test generation and test execution are implemented in different ways for different testing domains. A *testing domain* is defined by the type of SUT as well as by the testing objectives of the process. In a testing domain, it is not only the test generation process that is instantiated to suit different testing domains. For different kinds of systems, different properties need to be included in their models. Moreover, even when testing the same type of systems with different testing objectives (e.g., to reveal vulnerabilities, or to validate some behaviour against an asynchronous communication protocol) some properties need to be modelled while others can be disregarded. In this chapter, this thesis presents the particular modelling requirements as well as the generation process defined in detail for the (security) vulnerabilities, privacy and asynchronous systems testing domains.

4.1 Vulnerability testing

4.1.1 Modelling for testing vulnerabilities

Vulnerabilities in software systems usually result from (a series of) implementation choices, such as the use of existing components, the use of predefined structures available in programming languages, or the integration with other systems. In any case, a vulnerability is

a behaviour that was not specified originally but that is added "unwillingly" at some point during the implementation of the system.

The present testing framework uses three separate, but logically related, models for the testing of security vulnerabilities. The first model is a standard specification model that is usually abstract and is described as a state transition system. The second model applies the concept of fault injection and represents a localised mutation of the first model. It contains low level implementation details related to a specific security concern and usually is described as a program model. The third model represents the intentions of an attacker and works as a *test purpose* [118] for the test generation method used in this domain. The attacker's model contains unwanted behaviour of the system's environment. The attacker's model, implicit or explicitly, contains the data generation model for specific vulnerabilities. By separating the various models, the testing framework allows the specification to be incomplete or under-specified, and thus reduced in complexity, while it is still able to catch vulnerabilities using the implementation and attacker models.

The general description of the framework specified that a key concept on it is model composability. To facilitate the composition of the models, restrictions apply on how these models are specified. Consider $S = (Q^S, A^S, \rightarrow^S, {q^S}_0, F^S)$ being the abstract specification model and $I = (Q^I, A^I, \rightarrow^I, {q^I}_0, F^I)$ being the implementation model with V^I as the set of state variables in this model. Also consider $A = (Q^A, A^A, \rightarrow^A, {q^A}_0, F^A)$ being the attack model and V^A its set of state variables. Similarly consider that the abstract model states are also defined in terms of a set of state variables V^S. Under the previous considerations, for a well defined set of models, the following properties hold.

- The implementation model introduces concerns that were underspecified in the specification model, that is, there is a mapping function from V^S to V^I. This mapping is a total function.
- There is a mapping function from A^S to A^I that is also total.

The complexity of the attacker's model depends on the kind of vulnerability that it exploits. Simple attacks are modelled as unique transitions, e.g. an SQL injection attack on the

login function of a web based application. More complex attacks require a different number of preparation steps which lead the system to a vulnerable state but more importantly reveal information about the system. The attacker uses this information later to perform the attack.

Following [20], the framework assumes that the knowledge of the attacker is available. It is modelled within the state space of the attacker model and is denoted by K^A. The knowledge of the attacker records information about the structure and internal data of the software; i.e., database structure and users data. The attacker's model defines the special action *update_knowledge*() in A^I that updates K^A and a boolean symbol *known*() that returns *true* if a defined record in K^A has stored its correct value. Action *update_knowledge*() executes always when an action in A^I is executed. However it is not explicitly modelled unless the elements updated into K^A are of interest and enable the attacker to perform an attack.

This ends the definition of the modelling requirements for the security vulnerabilities testing domain. The models and elements described above are used in the test generation process described in the following section.

4.1.2 Generating tests to reveal vulnerabilities

For the domain of security vulnerabilities testing, this thesis suggests a fault-based approach to test case generation. Different from classical fault-based approaches, the approach in this thesis is not based on one faulty model, but in the combination of three models, as described in the modelling requirements for this domain. The implementation model is central to test case generation and identifies the implementation level vulnerabilities present in the software. When combined with the specification model it reveals under-specification problems in specific contexts. When combined with the attack model it gives an exact localisation of the security vulnerabilities.

The implementation model may not be an abstract model in a strict sense. There are situations in which the real implementation (or some of its modules or components) can be used as a "model". Consider, for example, a web application that uses a SQL engine in the background. It can be very complex to model an entire SQL engine. Furthermore, an abstract model of the SQL engine can fail to consider the same features that make the web application

vulnerable. Thus, in this case, it is recommendable to include a fully functional SQL engine as part of the "implementation model".

The combination of the three models produces what this thesis calls *faulty contexts*. From the conjunction of specification and implementation models this thesis identifies three faulty contexts mainly related to under-specification problems and not-holding assumptions. These contexts are illustrated in Figure 4.1.

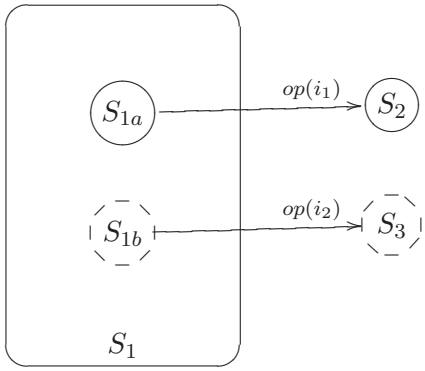
Figure 4.1(a) shows a context where there exists an under-specification in the input space. The initial state s_1 is divided into two different states s_{1a} and s_{1b}. The implementation of the operation *op* makes this distinction during execution. That is, the execution results in two different transitions with different final states. State s_3 is not described into the specification and is potentially harmful.

The context shown in Figure 4.1(b) refers to an under-specification problem in the output space. That is, the implementation contemplates a specific concern (represented by state or output variables) that is not considered in the specification. This new concern divides the final state s_2 into two states s_{2a} and s_{2b} where s_{2b} is a (potentially) dangerous state.

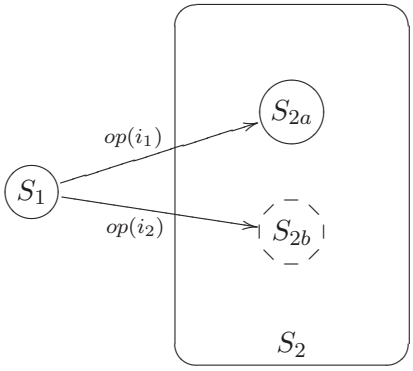
Finally, the context shown in Figure 4.1(c) describes the case where the specification assumes that the execution of operation *op* is triggered always from *safe* states s_1 and s_2. The specification does not consider dangerous state s_4 or assumes that this state is unreachable. However, the implementation contains a transition $s_4 \xrightarrow{op(i_n)} s_3$ that represents a security problem.

Considering the three models described in the modelling requirements, the test generation method described here consists of finding if a particular transition (defined by the attacker model) is present in the implementation model in any one of the faulty contexts described before.

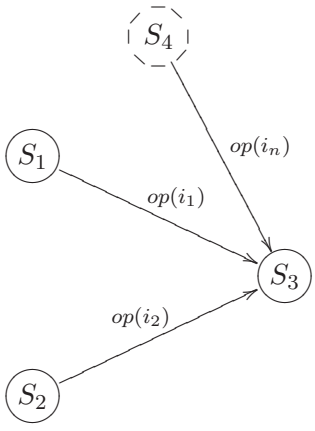
Automated analysis of the models is a desirable feature of any model-based testing framework. Automated analysis is also required to perform the search for a particular transition in all but very simple software models. Graphical representations are mainly useful for intuitive descriptions but can hardly be used in automated analysis. Therefore, in order to support tool based analyses we need to represent the models in a more suitable notation.



(a) Input-space under-specification



(b) Output-space under-specification



(c) Under-specified reachable states

Figure 4.1: Faulty contexts for test case generation

For practical purposes the testing framework in this particular domain uses first order logic predicates to represent the models. The translation process is straightforward as states in the transition systems can be represented as first order structures. With this translation as its aim, the testing framework defines state based expressions $Pre_{op}[v]$ and $Post_{op}[v']$ where v and v' represent the valuation of the state variables before and after op's execution, respectively. These logical expressions represent the pre- and post-conditions of the operation op.

With the pre- and post-conditions expressed as logical expressions the framework defines the existence of the transition $s \xrightarrow{op} s'$ in a model M as

- $Pre^M_{op}[v]$ is *true*; and
- $Post^M_{op}[v']$ is *true*

and, in general, the problem of searching for a defined transition is equivalent to finding a solution for the logical predicate $Pre^M_{op} \wedge Post^M_{op}$. As a notational convenience, hereafter references to these predicates drop the variables from the description when the pre-condition and post-condition involve all the variables of the relevant operation.

The faulty contexts previously described are defined also by logical predicates:

1. $Pre^S_{op} \wedge \neg Pre^I_{op} \wedge \neg Post^S_{op} \wedge Pre^A_{op} \wedge Post^A_{op}$

   that defines that the faulty transition executing op must start on an initial state of the specification (indicated by Pre^S_{op}) which can be accessed by the attacker (indicated by Pre^A_{op}) that is not considered valid in the implementation (indicated by $\neg Pre^I_{op}$). It should end on a final state different from the one expected in the specification (indicated by $\neg Post^S_{op}$) and this final state should be marked as a harmful state (indicated by $Post^A_{op}$). This captures the intuition in Figure 4.1(a).

2. $Pre^S_{op} \wedge Post^S_{op} \wedge Post^I_{op} \wedge Pre^A_{op} \wedge Post^A_{op}$

   defining that the faulty transition must start on a valid initial state of the specification (Pre^S_{op}) and must end on the expected state of the specification and of the implementation ($Post^S_{op} \wedge Post^I_{op}$). The final state (as described in the implementation model)

should be marked as harmful as it is part of the attacker model ($Post^A_{op}$). This state was marked considered safe ($Post^S_{op}$) in the specification. The faulty state was abstracted away while defining the 'super state' S_2 as shown in Figure 4.1(b).

3. $\neg Pre^S_{op} \wedge \neg Pre^I_{op} \wedge Pre^A_{op} \wedge Post^A_{op}$

   that defines that the faulty transition starts from a state not considered in the specification (unreachable state) or implementation (see Figure 4.1(c)) ($\neg Pre^S_{op} \wedge \neg Pre^I_{op}$) but available to the attacker (Pre^A_{op}). The transition must end in a state marked as a harmful state ($Post^A_{op}$).

If the generation process finds a solution for any of these predicates it means that it found a faulty transition in each of the correspondent faulty contexts. A faulty transition represents a negative test case (also called a counterexample) that if present in the implementation shows that the faulty model has been implemented. Given a standard test execution and verification process, the verdict for a negative test case must always be *fail* when executed in a correct implementation.

4.2 Privacy policies testing

4.2.1 Modelling for testing preservation of privacy

Privacy testing refers to testing if an implementation complies with a given privacy policy. The central element for this kind of testing is the privacy policy. However, policies are usually incomplete specifications and the framework requires an additional model that defines the general behaviour of the system. Thus, for privacy testing the specification is split into two models, the behavioural model and the privacy policy.

The behavioural model is described in terms of a LTS as it is usual in the present testing framework. However, this model contains enough information (implementation details) to be linked to the privacy policy. These implementation details turn this model into an implementation level model.

A behavioural model at the implementation level assumes the existence of sets *Var* and *Val* of variables and values, respectively. States on this behavioural model are represented as

a map from *Var* to *Val*, and actions are parameterised. A parameterised action, an element of $A \times Val$, $a(v)$ is present or contained in a behaviour $\mathcal{B} = (\sigma, \alpha, \omega, \rho)$ if α contains a and v is the correspondent value of a in ω. Henceforth, for this testing domain, this thesis refers to parameterised actions simply as actions, unless the contrary is explicitly stated. Additionally, for a variable x, the notation $\rho(x)$ stands for its current valuation and $\rho_q(x)$ stands for its valuation at the state q.

For privacy testing the testing framework defines, additionally, *Users* as the set of *user names* that represent all entities that can execute an *action* in the system. A user name is represented by a particular value, so *Users* is a subset of *Val*. Additionally, the framework defines that an action $a(v)$ is *executed over* a variable x if there exists a behaviour $b = (q \cdot q', a, v, \rho)$ in the model and $\rho(x) = v$. An *action execution* is always associated with a *user*. So the framework assumes that there exists a variable *currentuser* in *Var*, and for every state q there exist a value u in *Users* such that $\rho_q(currentuser) = u$.

Privacy requirements usually make reference to implementation level details. However, these requirements are usually written in natural language or in specific languages such as P3P[37] or EPAL[102]. To facilitate composability among the models, the framework requires a uniform and structured description of these policies. The framework refers to this structured representation as its *privacy modelling language*.

4.2.1.1 Privacy modelling language

The main components of the privacy modelling language are *rules* and *policies*. A *rule* is the most elementary unit of a *policy*. This is a generalisation of policy specification languages such as EPAL and P3P. In order to define *rules* and *policies*, variables and users of the system need to be categorised. Conceptually the categories are sets and a subset relationship defines a hierarchy among them. However, for a practical application, this thesis defines these types in a functional way.

- *DataCat* is a set of functions that categorise the variables of the system. For every f in *DataCat*, f is a function from $(Var \cup Val)$ to *Bool*, i.e., f's type is $f : (Var \cup Val) \rightarrow Bool$. The elements in *DataCat* form a partial order hierarchy such that if $data_1 \geq data_2$

then for all $v \in (Var \cup Val),\ data_2(v) \Rightarrow data_1(v)$.

- *AgentCat* defines a set of functions that categorise the *users* of the system. For every g in *DataCat*, g is a function from *Users* to *Bool*, i.e., g's type is $g : Users \rightarrow Bool$. The elements in *AgentCat* form a partial order hierarchy such that if $ag_1 \geq ag_2$ then for all $u \in Users,\ ag_2(u) \Rightarrow ag_1(u)$.

Privacy policies categorise users and data as several access-control policy languages do. Privacy policies are different, however, from other access-control policies, in that the former include into their rules the concept of *obligations*. An *obligation* is an action a system guarantees will be executed as a consequence of the execution of a current action. To be assessable, an obligation fulfilment needs to be bounded, usually, by the occurrence of an event. The following definition formalises the concept of an obligation and also states when it has been satisfied (fulfilled) by the system.

Definition 17 (Obligation) *An* obligation *is a triple* $O = (\beta, \delta, \upsilon)$ *where* δ *is an element of* DataCat*,* β *defines an action the system is committed to execute over a data element* y *such that* $\delta(y)$*, and the action* υ *defines when it can be evaluated if the obligation has been* satisfied *or not. Given an obligation* $O = (\beta, \delta, \upsilon)$*, let* q_β *and* q_υ *be the states of the system after the execution of actions* β *and* υ*, respectively. The obligation* O *is* satisfied *if there exist a behaviour* $\mathcal{B} = (\sigma, \alpha, \omega, \rho)$ *that contains* β *and* υ *and a predicate* p *such that* $p(q_\upsilon) \Rightarrow p(q_\beta)$*.*

Using the previous types and definitions, the following definitions formalise the concept of privacy rules and privacy policies.

Definition 18 (Privacy rule) *A privacy* rule *is a 6-tuple* (π, a, d, ag, c, O)*, where*

- π *:* {allow,deny} *stands for "permission" and defines either the rule* allows *or* denies *the execution of an action* $a(v)$ *: Action over a variable* $x \in Var$ *such that, given* d *: DataCat,* $d(x)$*;*
- *ag : AgentCat defines who is allowed or denied to trigger the execution;*
- *c is a predicate; and*

- *O defines an* obligation *the system acquires when the* rule *is applied.*

When the testing framework reads and analyses a privacy rule, it assumes the condition $c = true$ and the obligation $O = nil$ (where *nil* represents an empty record) whenever their values are not explicitly defined.

Definition 19 (Privacy policy) *A privacy* policy *is a tuple* $P = (AH, DH, R)$ *where AH is a set that contains the elements of AgentCat in the policy, DH is the set of all available elements of DataCat in the policy, and R is a sequence of rules.*

The order in which the *rules* are defined into the policy is important. Most policy specification languages rely on the order of their rules for conflict resolution. In the present framework this order defines applicability of the rules and, most importantly, prevents the generation of test cases that will lead to incorrect assessment. In our case, specific rules are written first while more general rules are written last. Given the rules $R = [r_1, r_2, \ldots, r_n]$ in a privacy policy P, the statement $r_i \leq r_j$ is true if and only if $i \leq j$. The operator $<$ between rules defines an order relationship which in turn is used in the generation process.

The collection of rules in a privacy policy is heterogeneous in the sense that some rules apply to specific parts of the system and not to others. Rules have also different scopes over the system, that is, some rules can be very specific and others can be more general. These characteristics define two relationships, a relationship between the rules and parts of the system that is referred to as *applicability* and another relationship among the rules themselves, referred to as the *conflict* relationship.

4.2.1.2 Applicability and conflict resolution

Privacy rules affect the workings of defined parts of the system but not others. Moreover, some rules are also intended to restrict some behaviours only when the system is on a previously specified state. From this arises the concept of *applicability* that defines when a rule has actually effect over the system's behaviour.

Definition 20 (Applicability) *Given a system represented by the model M and a privacy*

policy P, a rule $r = (\pi, a, d, ag, c, O)$ is applicable *if the system is in state $q \in Q$ and the following conditions are met:*

- *in the model M, $\exists a \in A \cdot \ q \xrightarrow{a(v)} q'$;*
- *action $a(v)$ is executed over $x \in Var$ and $d(x)$;*
- *$q(currentuser) = u$ and $ag(u)$ where $u \in Users$; and*
- *$c(q)$ holds.*

In any given state of the system, an *applicable* rule allows or denies the execution of action a and commits the system to fulfil the obligation $(\beta, \delta, \upsilon)$.

Definition 21 (Conflicting rules) *Whenever two given rules r_i and r_j are applicable in a given state q we say that r_i and r_j are in* conflict. *A privacy policy has a conflict resolution scheme such that, given the conflicting rules r_i and r_j, if $r_i \leq r_j$ then r_i is applied.*

The above definitions make it clear, given the current state of a system and an intended action, which rule from the policy applies to the system. This is useful when a system tries to enforce a policy. However, in a testing context, the inverse case arises. In other words, given a particular rule from the policy, the testing process needs to identify the state where the intended action of this rule will be applied. The problem arises when for a given rule r_j there exist other conflicting rules r_i, with $i < j$. In this case, the more specific rule r_i sets an exception to the more general rule r_j. Thus, a case for which r_j and r_i apply is not suitable for testing r_j. Nevertheless, intuitively if there is an exceptional case to a general rule, it means that there are other cases that will follow the general rule. The cases in which r_j does not conflict with r_i can be represented as a difference of rules $r_j - r_i$.

Definition 22 (Difference of rules) *The operation $r_k = r_j - r_i$ between rules generates a new rule r_k such that $\pi_k = \pi_j$, $O_k = O_j$, and:*

$$d_k = \begin{cases} data_j \wedge \neg data_i & \textit{if } data_i < data_j \\ data_k = data_j & \textit{otherwise} \end{cases}$$

$$ag_k = \begin{cases} ag_j \wedge \neg ag_i & \textit{if } ag_i < ag_j \\ ag_k = ag_j & \textit{otherwise} \end{cases}$$

$$c_k = \begin{cases} c_j \wedge \neg c_i & \textit{if } r_j \textit{ is in conflict with } r_i \\ c_k = c_j & \textit{otherwise} \end{cases}$$

Given a sequence of rules $[r_1, r_2 \dots r_n]$ we define the operation $r_k = r_n - [r_1, r_2 \dots r_(n-1)]$ as

$$r_k = \begin{cases} r_n - (r_{n-1} - [r_1 \dots r_{n-2}]) & \text{if } n \geq 2 \\ r_n & \text{otherwise} \end{cases}$$

This finalises the description of the models that specify the behaviour of the system and the properties for the preservation of privacy. In the following section, the definition of the test generation process uses these elements and concepts.

4.2.2 Generating tests for a privacy policy

In this testing domain, the actual behaviour of a system is usually specified by a behavioural model while the privacy requirements are specified as additional properties. These properties usually describe behaviours that are not allowed under predefined circumstances and other behaviours that are required, also under predefined circumstances. Thus, generated tests for this domain have to build a scenario where a specific behaviour is forbidden or required and then attempt to execute the actions that configure such behaviour.

Given a model M and a policy P, the test generation process follows Algorithm 1 to generate test cases for the present domain. Assume that for any rule r in the list of rules R, the condition c is represented in DNF, so that disjoint partitions can be easily identified. Additionally, after the calculation in line 2 all references to elements of a privacy rule in the

algorithm point to elements of r.

To simplify the discussion of the algorithm, consider that a behaviour is characterised by a string w. This string is a sequence of parameterised actions, so given a behaviour $\mathcal{B}_w = (\sigma_w, \alpha_w, \omega_w, \rho_w)$, w is the sequence of all $a_i(v_i)$, $a \in \alpha_w$ and $v \in \omega_w$, for $1 \leq i \leq n$ where n is the number of actions in α_w. A string w "reaches" a state q if $lastState(\mathcal{B}_w) = q$. This framework assumes that behaviours with values are always deterministic, this is, values resolve any non-determinism present in abstract models. Therefore, the behaviour $\mathcal{B}_w$ is constructed from the definition of w.

The generation algorithm looks into each of the rules in the privacy policy, calculates any exception to the current rule due to previous rules being applied and generates a new rule. Then it tries to break this rule by generating test cases that execute a forbidden action, setting the oracle to *false*. This indicates that a correct implementation that complies with the policy should fail on these test cases. On the other hand, it also generates test cases where an action has an associated obligation. These test cases pass after the obligation has been fulfilled. A system that complies with the policy should pass these test cases.

Algorithm 1 Test case generation

1: **for all** rules $r_i \in P$ **do**
2: calculate r **:=** $r_i - [r_1 \dots r_{i-1}]$
3: **for all** states $q_i \in Q$ such that r is applicable in q_i **do**
4: **for all** disjoint partitions c_n of c **do**
5: generate a string w_n such that $\mathcal{B}_{w_n}$ *enables* a and $c_n(lastState(\mathcal{B}_{w_n}))$
6: **end for**
7: **for all** strings w_n **do**
8: **if** $\rho = deny$ **then**
9: let $w = w_n\ a$, add the test case ($\mathcal{B}_w$,*false*) to the test suite
10: **if** $O \neq nil$ **then**
11: let wp be a string such that $\mathcal{B}_{wp}$ *enables* υ
12: let $wo = w\ wp\ \upsilon$
13: let q_β be a state such that $\exists\, q \cdot q \xrightarrow{\beta} q_\beta$
14: let p be a predicate such that $p(q_\beta)$
15: add a test case ($\mathcal{B}_{wo}$,p) to the test suite
16: **end if**
17: **else if** $\pi = allow$ and $O \neq nil$ **then**
18: let wp be a string that *enables* υ and $w = w_n\ a\ wp\ \upsilon$
19: let q_β be a state such that $\exists\, q \cdot q \xrightarrow{\beta} q_\beta$
20: let p be a predicate such that $p(q_\beta)$
21: add a test case ($\mathcal{B}_w$,p) to the test suite
22: **end if**
23: **end for**
24: **end for**
25: **end for**

The test case generation algorithm assumes that υ is a user-triggered action. Thus, it does not generate test cases where an obligation will *eventually* be fulfilled, but cases in which the obligation is fulfilled in a bounded future defined by a finite sequence of actions.

Table 4.1: Example of privacy rules

						O		
rule	π	a	d	ag	c	β	δ	υ
r_1	deny	b	dataRoot	agentRoot	-	-	-	-
r_2	allow	a	dataRoot	agentRoot	-	c	dataRoot	d

Figure 4.2 shows an extremely simple system model that serves to explain the functioning of the algorithm. Consider also the privacy rules shown in Table 4.1 where *dataRoot* and *agentRoot* represent the categories in the top of hierarchies *DataCat* and *AgentCat* respectively. Rule r_1 forbids the execution of action b in any circumstance and rule r_2 allows action a to be executed but committing the system to also execute c before d is executed.

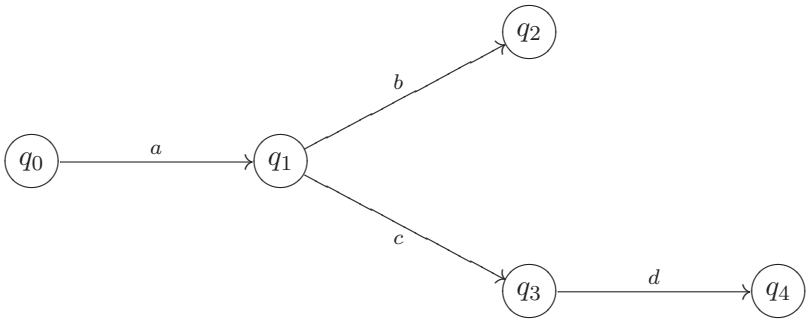


Figure 4.2: A simple system model.

When the algorithm is applied to generate test cases, the calculation in line 2 does not modify the rules, that is, the rules are independent of each other and the algorithm processes them separately. Processing rule r_1 the algorithm identifies state q_1 as the only one to enable action b. Then it generates string $w_1 = a$ that reaches state q_1. Rule r_1 exercises only the part of the algorithm dealing with denying actions, lines 8 to 15. Thus, it generates a string $w = a \cdot b$ containing the enabling string w_1 and the forbidden action b. Any string containing the forbidden action should fail to have a matching run in a correct implementation. In this case, test case $(\mathcal{B}_w, \mathit{false})$ is generated. The algorithm then skips lines 10 to 15 and ends because there is no obligation ($O = nil$) to be processed.

Now, rule r_2 is processed. The algorithm identifies q_0 that enables action a. Then, enabling action w_1 is set to the empty string. Rule r_2 exercises only the part dealing with allowed actions with obligations, lines 16 to 21. Then, string $w = a \cdot c \cdot d$ is generated, with the

sub-string $wp = c$ being the enabling string for action d. From the composition of the string w we know that a successful run of w will execute action c, thus, satisfying the obligation. Finally, the test case $(\mathcal{B}_w, true)$ is generated, with the predicate *true* indicating a successfully execution. As there is only one system level behaviour only one test case per rule is generated.

4.3 Asynchronous systems testing

The models used in the testing domains of security vulnerabilities and privacy policies describe the systems in a centralised way. In other words, they consider the system as a monolithic and unique entity. Even when a system could be built out of several distinct components, these models consider that the actions are executed by the system as a whole and don't differentiate between particular components.

The previously described models also assume that actions are executed sequentially in an predefined order. In practical terms, when several components of the system interact, the models assume that the components of the system know when it is their turn to execute an action and they wait while other components are executing their correspondent actions. In summary, these models describe systems from a global perspective where all the components of the system (if more than one) execute their actions synchronously.

There are however systems in which the components execute concurrently in an asynchronous way. Moreover, there are systems for which one component executes (or seems to execute) concurrently different instances of itself. For these kinds of systems, it is not practical to carry on their testing from a global perspective. For them, testing is focused on one (or more) defined component(s) of the system while other components become part of the environment.

4.3.1 A different kind of system: Example

Consider a system of file exchanging where a user (the sender) connects with another one (the receiver) and sends a file. There are, obviously, two distinguishable sides on this system. From the sender's point of view, his side of the system is responsible for executing only a subset

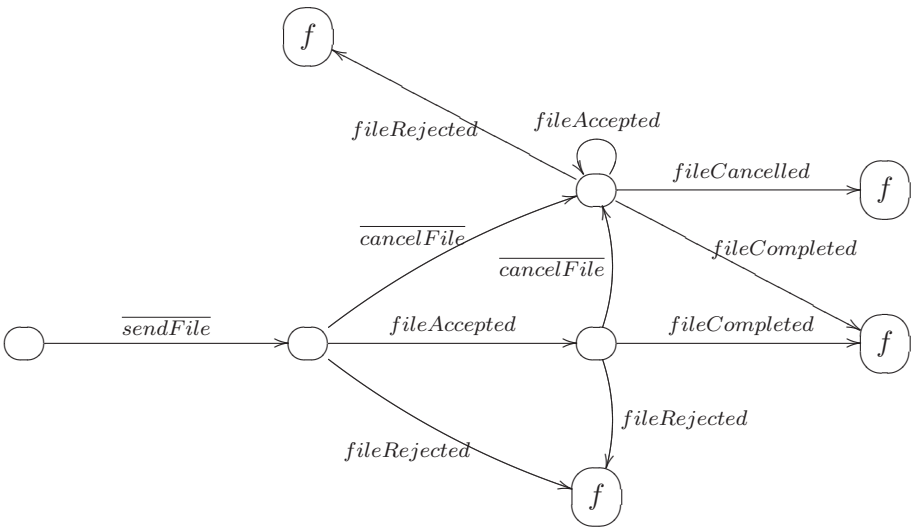


Figure 4.3: Simplified model of a file exchanging system

of the available actions. The sender calls them, the subset of *controllable actions*. The other subset of actions is executed by the receiver. However, the sender needs to be notified when the receiver executes one of these actions. Thus, the sender calls those actions the subset of *observable actions*. This example describes the actions that take place in this system mainly from the point of view of the sender.

A user can initiate the process by executing the action *sendFile*. The receiver can, then, accept the file or reject the file. From the sender's point of view, when the receiver accepts the file, action *fileAccepted* is observed. On the other hand, if the file is rejected, action *fileRejected* is observed. Even when the receiver has accepted the file, the transfer can be aborted at any time before it is completed. An aborted transfer results also in the occurrence of action *fileRejected*. The sender can also cancel the transfer at any time, which is represented as the execution of action *cancelFile*. Finally, if an accepted file is successfully transferred, action *fileCompleted* is observed. The graph in Figure 4.3 shows a pictorial description of the interactions of this system. Controllable actions are denoted $\overline{sendFile}$ and $\overline{cancelFile}$ for better readability. Accepting states are also labelled with an f. Two states with the label f are not necessarily the same state.

This system is asynchronous because controllable and observable actions can happen in any order. For example, consider the case where a user is sending a file and before it receives

any response, decides to cancel the operation. At the same time, the receiver observes the requests and rejects the file. From the sender's point of view the user either observes that the cancel request was processed or that the receiver has rejected the file. Any of these two cases is acceptable to the sender in the knowledge that if the file is rejected the cancel request will never be processed. However, it can also be the case that the cancel request arrives after the receiver accepted the transfer and it was completed. In this case the cancel request is just discarded.

Asynchronous behaviour appears also because the sender does not need to wait for the receiver to accept or reject a file. The sender can send a file and then another one before he observes any action executed by the receiver. Moreover, when two or more files are sent, the sender will expect different confirmation or rejection messages, one for each file sent, and the system has to be able to differentiate to which file each message "belongs". Such behaviour is represented using LTS by interleaving the actions related to each of the files being transferred. However, this "interleaved" LTS model needs to provide a way to differentiate between different executions of the same action. Additionally, there is no guarantee about the order in which observable messages will appear and thus, all possible combinations need to be considered.

4.3.2 Modelling asynchronous systems

Systems are called asynchronous because they perform asynchronous communication between two or more of their components. These components refer to different elements, from concurrent elements that execute the same code to distributed elements in networked architectures. In an abstract way, this testing framework assumes that there exist two parts in an asynchronous system. There is one part whose actions are controlled by the testing framework and there is another part whose actions are only observed by the testing framework.

The present testing framework uses LTS for representing asynchronous systems. To distinguish between different executions, actions in A are parametrised. This is, actions in A are denoted $a(id)$ where a is the action name and id is a special variable such that different values of id represent different executions of a. There are several sets of values which can

be assigned to *id* for distinguishing between different action executions, e.g., timestamps, sequential id's. These values depend on the implementation and are discussed in the case studies. In the general discussions, however, this thesis refers to actions only by their names unless it is necessary to include *id*'s value.

In these LTS models, the set A of actions is partitioned into A_c and A_o disjoint subsets of controllable and observable actions, respectively. As mentioned in the description of the example, the testing framework calls the set of actions that can be directly invoked by the testing environment, *controllable actions*. Contrarily, it calls the set of actions that are executed independently by the SUT (or its environment), *observable actions*.

The LTS model of an asynchronous system shows all the possible interleavings of controllable and observable actions. The present testing framework places one restriction to the kind of models for which it can generate test cases. This restriction is called the *asynchronous property* and defined as follows.

Definition 23 (Asynchronous property) *Given an observable action a_o and q the current state of the system, a model $M = (Q, A, \rightarrow, q_0)$ is considered to be asynchronous if* enabled(q, a_o) *implies that*

for all q_j such that $q \xrightarrow{a_c} q_j$ and $a_c \in A_c$, enabled(q_j, a_o)

The asynchronous property establishes that in the model of an asynchronous system, once an observable action is enabled at some point in the execution, it can be "disabled" only by another observable action. This effectively means that the model assumes that the state of the "observable components" of the system and the environment does not change until these components show an observable behaviour.

Finally, to improve readability in the following sections, given a behaviour $\mathcal{B}_M = (\sigma q, \alpha, \omega, \rho)$, denote *observableSuccessors($\mathcal{B}$)* the set of enabled observable actions in state q, defined as $observableSuccessors(\mathcal{B}) = \{a | a \in A_o \wedge \exists q',\ q \xrightarrow{a} q'\}$.

4.3.3 Testing asynchronous systems

In the testing domain of asynchronous systems, the framework needs to specialise its definition of test cases. The general description of the test generation process defined a test case composed by a behaviour and a state predicate. However, this definition is not suitable because it does not consider the case where different sequences of actions represent the same behaviour. This case is common in asynchronous systems where the order in which actions are observed do not necessarily represent the order in which they were executed. For this case the testing framework needs to extend its definition so that a test case is composed by a set of possible behaviours and a state predicate.

In order to define a test case in a more formal way, given a sequence $\alpha \in A^*$ and $A_s \subseteq A$ this framework introduces $w \downarrow_{A_s}$ as the sub-word obtained by erasing all the symbols not in A_s. Additionally, given a behaviour $\mathcal{B}$, the framework denotes also $\alpha_{\mathcal{B}}$ the sequence of actions α in the behaviour $\mathcal{B}$. Now, the testing framework formally defines a test case for asynchronous testing.

Definition 24 (Test case) *Given a sequence of controllable actions $C = a_{c1} \cdot a_{c2} \cdot \ldots \cdot a_{cn}$ and the set of behaviours B_M defined by a model M, a test case is a pair $TC : (B_C, SP)$ where*

- *B_C is the subset of n behaviours represented by $\{\mathcal{B} | \mathcal{B} \in B_M \ \wedge \ \alpha_{\mathcal{B}} \downarrow_{A_c} = C \ \wedge \ \forall \mathcal{B} \in B_C$, $lastState(\sigma_{\mathcal{B}}) \in Q_F$, and*
- *sp is a state predicate such that for the last state of each $\mathcal{B}_i$ in B_C, $sp_i(lastState(\sigma_{\mathcal{B}}))$ holds.*

The previous definition states two properties for a generated test case for asynchronous systems. These properties are used to determine when a test case passes or fails its execution. These properties are

1. for all sequences of actions in a given test case, controllable actions are always executed in the same order; and,

2. for all sequences of states in a given test case, the last state will always be an accepting state.

The first property states what is common for testing theories based on abstract models, which is, a test case *passes* if the implementation can execute the same (sub)sequence of (controllable) actions as the model. The second property relies on the definition of accepting states to determine whether a generated set of behaviours can be considered a test case. As an example, a common definition of accepting states for a wide range of protocols assume that the execution of observable actions is triggered by controllable actions. Therefore, expected observable actions represent pending responses of the system whereas accepting states are those in which no responses are pending. Then, for this kind of systems, the second property states that for a test case to pass, all the required responses should have been received.

To deal with asynchronous testing the testing framework needs to execute asynchronously with respect to the observable elements of the system. In other words, to be capable of producing different interleavings of controllable and observable actions, just as the real system does, the testing framework needs to be able to observe actions executed by the environment and to execute controllable actions concurrently. Thus, for this testing domain, the framework defines two algorithms, the test generator and the execution observer, which run in parallel during the testing process. Both execute independently but communicate (and synchronise if necessary) by accessing two global structures, the communication channel CC represented as a queue of executed actions and a set S_o of expected observable actions.

An important assumption about the communication channel is that it is a perfect channel, without losses or delays. This means that by using this channel, the testing framework is synchronised with the SUT (the controllable components of the system). A change in the SUT's state is reflected instantaneously in the model's state. Similarly, requests to execute controllable actions arrive at the SUT as soon as the tester (a human tester or an automated process that acts as a tester) decides to execute a defined action.

Algorithms 2 and 3 show the generation and observation processes, respectively. In the algorithms, the behaviour B represents the current generated behaviour for the test case TC, q represents the current state of the system in the model and S_o denotes the current

set of *observable successors*. The algorithms also assume that all messages from the SUT notifying of the execution of an action are observed in the communication channel CC. The function $extend(B, a)$ takes a current behaviour B and an action a as parameters and returns a extended behaviour. Additionally, function *front(Q)* returns the element at the front of queue Q without removing it and function *dequeue(Q)* removes the element at the front of the queue Q and returns it to be used in the algorithms.

Algorithm 2 Test generator

1: $B = (q_0, \epsilon, \epsilon, \rho_0)$
2: $TC = (B, true)$
3: $q = q_0$
4: **while** S_o is not empty **do**
5: **while** $\exists a_c \in A_c$ such that $enabled(q, a_c)$ **do**
6: **pick** an enabled a_c
7: send a_c to the SUT
8: wait until $front(CC) = a_c$
9: m **:=**dequeue(CC)
10: B**:=**$extend(B, m)$
11: TC**:=**$(B, true)$
12: update current state q
13: S_o**:=**$observableSuccessors(q)$
14: **end while**
15: **end while**
16: **if** $q \notin F$ **then**
17: TC**:=**$(B, false)$
18: **end if**

Algorithm 3 Execution observer

```
1: while Q_c is not empty OR S_o is not empty do
2:   if front(CC) ∈ A_o then
3:     m:=dequeue(CC)
4:     if m ∈ S_o then
5:       B:=extend(B, m)
6:       TC:=(B, true)
7:       update current state q
8:       S_o:=observableSuccessors(q)
9:     else
10:      Test fails
11:      B:=extend(B, m)
12:      TC:=(B, false)
13:      exit while
14:    end if
15:  end if
16: end while
```

Summarising the process, the test generator appends actions to be executed to the queue Q_c, therefore it also appends expected observable actions to the set S_o which, in principle, plays the role of test oracle for the test case. The execution observer removes the actions from Q_c or S_o as the execution is performed by the implementation. However, as non-determinism is resolved at certain points by implementation choices, the set of expected actions S_o needs to be modified to reflect the choices. Then, the choice of an observable action a_o will remove additional actions from S_o which will not be executed after a_o. Naturally, a_o's observable successors will also append new actions to S_o.

To complete the testing process, the execution of a test case needs to produce a verdict. The algorithms do provide the concept of a test case's *fail* verdict. From this concept the framework derives the concept of a *pass* verdict. A test case passes if its execution does not fail and the algorithms terminate. An *inconclusive* verdict is generated when the algorithms

do not terminate. However, the tester can force the *fail* verdict in those cases by introducing proper *time-out* events.

4.4 Concluding remarks

This chapter has presented the specialisation of the general testing framework for the testing domains of security vulnerabilities, privacy and asynchronous systems. Up to three elements of the general framework were specialised; the models, the test generation algorithm and the test execution algorithm, depending on the specific characteristics of the testing domain.

First, the specialisation of models reflected the specific characteristics of the systems that needed to be modelled. For the domain of security vulnerabilities, the specialised models were logically separated into three, namely the specification model, the implementation model and the attacker's model. This separation was necessary because of the need to describe the specific properties of an implementation that make it vulnerable, as well as the objectives of a malicious user. For testing privacy policies the general behavioural and data generation models were complemented with a privacy model that represented the privacy requirements. A general way of representing this privacy model was defined which is independent of current privacy languages. Finally, for modelling asynchronous systems it was necessary to partition the set of actions into controllable and observable actions. For the kind of models considered in this thesis an *asynchronous property* was defined such that whenever an observable action is enabled by a controllable action, it cannot be disabled by subsequent controllable actions. This property was needed to allow the reorder of observable actions from the tester's perspective.

In second place, the test generation algorithm was specialised to reflect the testing objectives for each domain. For testing privacy policies and asynchronous systems, specific algorithms were presented. However, no specialised algorithm was needed for vulnerabilities testing. In this domain, the objective of the testing process turned out to be that the malicious user cannot reach his objectives under the described scenario, that is, by exploiting the modelled properties in the way that was described in the attacker's model. Instead of

developing a specialised algorithm for the generation process, the models were translated to first order logic predicates and a standard constraint solving algorithm was applied. Once a solution was found, standard sequencing and execution algorithms were applied.

Finally, the only specialised algorithm for test execution was developed for the domain of asynchronous system testing because it needed to execute and observe actions in parallel. It was defined that the suitable way of generating and executing a test case for this domain was to use an online approach.

The next chapter presents the application of the general approach as well as the specialised ones to several case studies.

Chapter 5

Case studies

5.1 Overview

This chapter presents four case studies with the aim of demonstrating the applicability of the general and specialised approaches presented in the previous two chapters. All the case studies also aim to demonstrate via examples how the framework presented in this thesis can be implemented using tools and other artefacts already developed for current test automation frameworks. With this objective in sight, the present studies use mainly the SmartMBT tool (described previously in Chapter 2) to drive the generation of test cases and to perform their execution. Enhancements to this tool have been developed where necessary and are described in this chapter. Complementary tools or frameworks are also used when necessary and are properly described.

The four case studies that are discussed in this chapter are

Device drivers testing. This case study demonstrates the application of the general framework to the testing domain of operating systems. The aim of this study is to show the advantages of using a model-based testing framework in terms of greater fine-control and diversity of the test scenarios. It is also an objective of this study to show how a model-based framework can be implemented so that it reuses elements of existing testing frameworks.

Vulnerabilities testing. In this case study the applied framework is specialised for the

generation of test cases that reveal vulnerabilities. Two web applications serve the purpose of demonstrating the generation process. A first exercise is focused on the generation of test cases and a second one on the sequencing and execution of the test cases. In addition to the model-based testing tool, a customised constraint solver is built and used in the generation process.

Privacy testing. This study shows the applicability of the specialised framework for testing privacy policies. Two different privacy policies are used for the demonstration. A first exercise focuses on the generation and execution of test cases for testing how web browsers guard the privacy of the user's navigation information. A second exercise demonstrates the generation of test cases for a privacy policy that includes obligations.

Asynchronous systems testing. In this case study, the specialised framework for testing asynchronous systems is applied to an open implementation of the Financial Information eXchange (FIX) protocol. This study uses a simplified model of the standard specification for the FIX protocol. The tester controls the Client module of the open implementation and observes the responses of the Server module. This study uses an on-line approach for generating and executing test cases.

5.2 Device drivers testing

Testing operating system's kernels and modules is difficult due to the complexity of such pieces of software [70]. They usually include code developed by several different programmers, in different places and time. Moreover, due to their role in a system their code usually includes references to "low level" elements such as the hardware components of the system.

Most of what is publicly known about the development and testing of operating systems comes from the study and observation of the development of open (source) systems, starting with Linux and more recently, for example, OpenSolaris. Historically, until early in the 2000s, Linux testing efforts were primarily informal and ad-hoc in nature [75]. The major testing effort in the development of these systems was usually performed by the developers themselves. However, currently, developers of open systems are usually provided with a suite

of tests which needs to be applied against the code they are developing. These test suites are used as a way to ensure that the code contributed by different programmers maintains standard interfaces and does not break other functionalities.

Test suites are software packages written for the express purpose of testing. They usually cover a wide range of functions and they are written to expose flaws that are likely to be in the system. In the case of operating systems, these test suites are composed of scripts that perform shell invocations of the operating system commands. Examples of these test suites are the ones that the OpenSolaris community* has developed with the aim of testing the implementation of different modules of the operating system, storage device drivers being among them.

In this case study we refer to the particular case of a test suite for the HBA (Host Bus Adapter) storage driver of the OpenSolaris operating system. The aim here is to show how a model-based testing approach can bring two particular capabilities to the current test suites for the HBA driver. These capabilities are

- the ability to execute different test scenarios without having to modify their source code each time a new scenario is required, and
- the ability to invoke fine-grained control over the operations that the OS executes.

This study refers to these capabilities as the *flexibility* of the test suite.

5.2.1 The system under test

The Host Bus Adapter (HBA) provides input/output processing and physical connectivity with storage devices such as hard disks. A driver that controls this adapter in an operating system (OS) provides several services to the other OS modules and applications. A non-exhaustive list of the operations that this driver performs includes

- format a device,
- partition the device with a variable size,

*http://www.opensolaris.org/os/community

- convert a partition into a file system,

- mount a file system, and

- create files into a file system.

The list above is restricted to the management of file systems inside a storage device, a device is a physical element of the system, typically a hard disk. The format of a device defines how the structure of this device is represented. In OpenSolaris, two different formats are available, the SMI and the EFI formats. They differ mainly in how many partitions (also called slices in OpenSolaris) the device can contain and how they are referred to by the other OS modules. Each partition represents some physical space in the device. Typically, partitions in hard disks have a defined number of cylinders which defines their size. Inside a partition, the operating system and other applications store files.

A file system defines the way to refer to and organise files inside a storage device. Different file systems are available in OpenSolaris: FAT32, UFS and ZFS among them. File systems are accessed by mounting them. A file system is mounted into a directory called the mount point. File systems and mount points are in a one-to-many relationship. That is, a particular mount point provides access to a unique file system. However, different mount points can also provide access to the same file system.

The list of operations for the HBA driver presented at the beginning of the present section will be useful at the time of defining a model for this driver. Hereafter, general allusions to the HBA driver refer particularly to these functions and elements, unless the contrary is explicitly stated. The remainder of this section describes the framework in which the current test suite for the HBA driver is based.

Current testing framework

The current tests for OpenSolaris storage management are contained in the Storage Driver Test Suite (SDTS)[†] and use the Test Environment Toolkit (TET) as the underlying testing framework. The structure of these tests is shown in Figure 5.1. Each *test suite* is made up

[†]http://www.opensolaris.org/os/community/storage/tests/

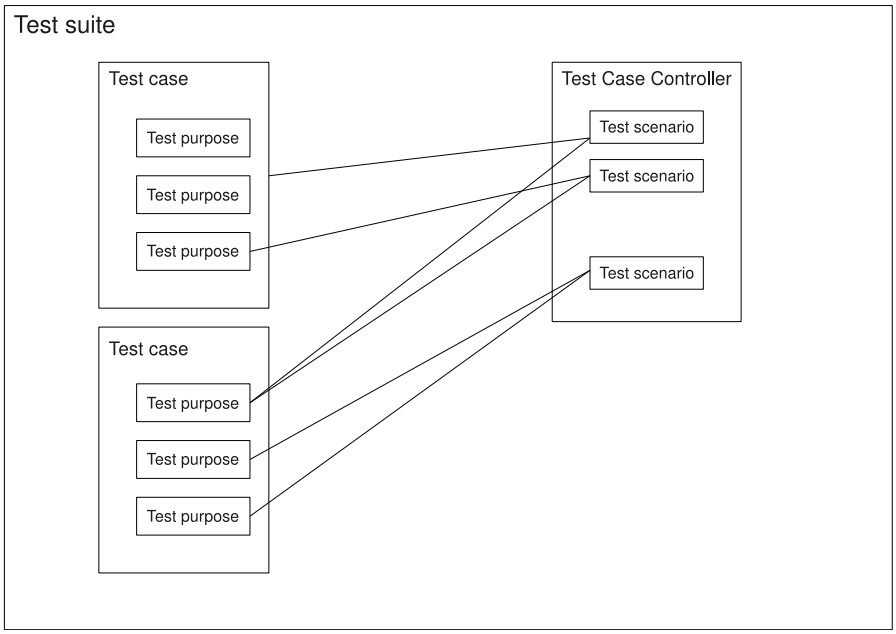


Figure 5.1: Structure of test suites in the TET framework

of one or more *test cases*. Each test case is an executable program constructed by grouping together test functions, called *test purposes*. In general, a test case defines the order of execution of its test purposes. The Test Case Controller (TCC) processes the test cases and executes the corresponding test purposes. A *test scenario* file specifies the list of test cases that the TCC processes.

The code in Figure 5.2 shows an extract of a *test scenario* file. This code specifies different scenarios available for execution. For example, if the tests are invoked for execution without specifying a particular scenario, the scenario *all* will take place and all test cases will be executed. However, if the scenario *format_part* is invoked then only test purposes 6, 8 and 9 of the test case *tc_format* will be executed. The test cases considered in this scenario file correspond to tests for different uses of the *fdisk* command, the *format* command and the creation of new file systems that involves the previous and other commands of the operating system.

```
all
  ^fdisk
  ^format_all
  ^newfs

fdisk
/tests/diskcmd/fdisk/tc_fdisk

format_all
/tests/diskcmd/format/tc_format

format_part
/tests/diskcmd/format/tc_format{6,8,9}

newfs
/tests/diskcmd/newfs/tc_newfs
```

Figure 5.2: Test scenario specification

When writing test cases in the TET framework, the tester is required to supply the test purpose code that actually executes the test operation. This framework is modular so that the code that implements functionalities common to more than one test purpose is written as library functions and called from inside the test purposes. As an example, Figure 5.3 partially shows the code of a test purpose included in the SDTS. This code invokes library functions *label_smi* and *build_ufs* (shown in Figures 5.4 and 5.5)and traps the results of their execution. Then, it logs these results for reporting purposes. Functions *label_smi* and *build_ufs* are part of a repository of common utility functions. The function *label_smi* formats a previously identified disk (defined in variable $stds_disk) and builds an SMI label into it. The function *build_ufs* creates a partition into the disk, allocates a new file system into it and mounts the created file system into a predefined directory $mp.

An analysis of the code in the test purposes as well as the code in the common library functions shows that the test purposes on this suite are written at a high level of abstraction. In other words, the scripts do not correspond to the invocation of simple shell commands but to a structured sequence of invocations. This, in principle, reduces the flexibility of the test suite because operations are only considered inside a predefined sequence and cannot be tested in a different context without rewriting or adding more scripts.

```
  ⋮
if ! label_smi >> $logfile 2>&1; then
cti_report "FAIL: label smi on $sdts_disk"
cti_reportfile "$logfile"
cti_fail "tp_newfs_001: FAIL"
return
else
cti_report "PASS: label smi on $sdts_disk"
fi
  ⋮
if ! build_ufs >> $logfile 2>&1; then
cti_report "FAIL: create ufs on $rdev and mount it to $mp"
cti_reportfile "$logfile"
cti_fail "tp_newfs_001: FAIL"
return
else
cti_report "PASS: create ufs on $rdev and mount it to $mp"
fi
  ⋮
```

Figure 5.3: Test purpose code

Approaches that consider tests with a strictly predefined sequence of operations, as the one presented here, usually require the system to be in a predefined suitable state before the test sequence can be executed. This is the case of the SDTS suite where test purposes have been designed to be executed independently. Therefore, each test purpose modifies the initial state of the system to enable its execution and they must clean the state of the system so that the next test purpose can be executed without any problems. As a result, the test case defines explicitly the order of execution but the results of one execution do not influence the results of subsequent executions. This does not represent a real usage scenario. However, this characteristic is useful in trying different scenarios by reordering the execution of test purposes without having to modify their code.

5.2.2 Test generation with models based in the SDTS suite

This case study implements a model-based testing framework on top of the existing framework with the aim of providing the current test suites with the ability to execute more and different test scenarios without having to modify their existing code. In order to achieve this, a model of the current test suite takes care of driving the testing process and generating new test

```
function label_smi
{
   if ! print "label\n0\n\n\n\nq" > $testdir/label_smi.txt; then
   echo "create file label_smi.txt failed"
     return 1
    elif [[ ! -f $testdir/label_smi.txt ]]; then
      echo "label_smi.txt doesn't exist"
     return 1
    fi

    if is_i386; then
     if ! fdisk -B /dev/rdsk/$sdts_diskp0; then
     if is_less_1tb $sdts_disk; then
     echo "The size of the disk is less than 1TB"
     echo "label smi failed"
     else
     echo "The size of the disk is more than 1TB"
     echo "label smi failed"
     fi
       return 1
     fi
     if ! fmthard -s /dev/null /dev/rdsk/$sdts_diskp0 > /dev/null 2>&1; then
     echo "create partition 2 entry size equal to the full size of the disk failed"
     fi
    fi

    if ! format -e -f $testdir/label_smi.txt -s $sdts_disk; then
     if is_less_1tb $sdts_disk; then
     echo "The size of the disk is less than 1TB"
     echo "label smi failed"
     else
     echo "The size of the disk is more than 1TB"
     echo "label smi failed"
     fi
     return 1
    fi

    return 0
}
```

Figure 5.4: Library function *label_smi* code

```
function build_ufs
{
  if [[ ! -d $mp ]]; then
    mkdir -p $mp
  fi

  if ! make_mp_available; then
    echo "$mp is unavailable"
    return 1
  fi

  if yes | newfs $rdev >/dev/null 2>&1; then
    echo "using newfs create ufs on $rdev successfully"
  else
    echo "using newfs create ufs on $rdev failed"
    return 1
  fi

  if mount $bdev $mp >/dev/null 2>&1; then
    echo "mount slice $bdev to $mp successfully"
  else
    echo "mount slice $bdev to $mp failed"
    return 1
  fi

  return 0
}
```

Figure 5.5: Library function *build_fs* code

Table 5.1: States of a model for the SDTS suite

state	availableDisk	is386	less1Tb	SMILabel
s_1	true	true	true	true
s_2	true	true	true	false
s_3	true	true	false	true
s_4	true	true	false	false
s_5	true	false	true	true
s_6	true	false	true	false
s_7	true	false	false	true
s_8	true	false	false	false

scenarios on the fly.

5.2.2.1 Modelling based on the SDTS suite

In the testing framework that this thesis defines, a model contains mainly two elements, states and action triggered transitions. This study concentrates on the reuse of the current set of scripts in the SDTS suite. Therefore, an analysis of the current code defines the proper states and transitions. There are four boolean functions that define the state of the system and, thus, define if and when a particular test case (or purpose) can be executed:

- *is386* that indicates if the system architecture is 386 compatible or not,
- *less1Tb* that indicates if the disk's capacity is less than 1 Tb,
- *availableDisk* that returns the disk's availability,
- *SMILabel* that indicates if the disk can have a SMI label attached.

These four functions are part of the library of common functions for the test suite. In principle, they define sixteen states for the system under test. However, states where there is no disk available for the tests do not lead to a useful test. Therefore, the present model considers only eight states - namely those where there is an available disk for the tests. The states for this model of the system are described as in Table 5.1.

Just as with the states, actions for the model also derive from the scripts of the test purposes of the SDTS suite. In this case actions match one-to-one to the test purposes of the

Figure 5.6: Model of the SDTS suite

suite. For example, actions *fdisk1*, *fdisk2*, *fdisk3* and *fdisk4*, derive from the test purposes in the test case *fdisk*: *fdisk001* up to *fdisk004*. The same applies for test cases *format*, *newfs* and *vtoc*.

With states and actions a LTS model that represents the SDTS suite can be described. Figure 5.6 shows a simplified pictorial description of the model for the SDTS suite. For clarity, this description uses the test case name to label the transitions and it stands for all the actions related to that test case. For example, *format* stands for actions *format1*, *format2*, up to action *format10*.

Figure 5.6 shows only two states of the SDTS model, s_1 and s_2, mainly because of two reasons. In the first place, it is because all states from s_2 to s_8 present exactly the same behaviour, that is, actions (and the test purposes they represent) *fdisk*, *format* and *newfs* can be executed in any of these states. Therefore, state s_2 is sufficient to represent pictorially all these states. However, state s_1 is the only state in which the test purposes of test case *fdisk* can be executed. Secondly, considering only the actions included in the test suites, the states are independent one from the other. In other words, there are no transitions between the states. This is because the state of the system is defined in such terms that none of the actions can trigger a change of state. This means that, for example, if the testing process is carried on a system with SPARC architecture there is no way that the system can change its architecture to a 386-compatible architecture during this process.

5.2.2.2 Test generation

Test cases are generated from the model of the system with the main objective of producing as many different case scenarios as possible. As indicated before, this study uses the SmartMBT tool to drive the generation process. This particular exercise in the present case

study applies a random approach to the test generation. This approach provides flexibility to the test generation process. In fact, different executions of the process using this approach produce different test sequences which normally lead to different scenarios. This approach also interleaves test purposes from different test cases into the test sequence. This characteristic of the test sequence is important during test execution because it challenges assumptions such as the premise that every script cleans the system state after execution.

For this exercise, the generated test sequence includes the following subsequence of 10 actions: *format1*, *vtoc1*, *format3*, *newfs1*, *format1*, *newfs3*, *fdisk3*, *format6*, *format7*, *newfs2*. The execution of this subsequence is discussed in the following section.

5.2.2.3 Test execution

The model-based framework executes the generated test cases via the implementation of the Adaptor module. Generally, the Adaptor uses a set of scripts that perform (or call) the operations in the SUT. In this case, given that each action maps to an existing script, the main function of the Adaptor is just to execute the appropriate script in the SDTS suite. As indicated before, this study uses the SmartMBT tool to trigger the execution of test cases. The Adaptor, therefore, is implemented as a service that listens for requests of the SmartMBT tool and links them to the appropriate script.

The other function of the Adaptor is to capture the responses and the resulting state of the system after the execution of the scripts so that a verdict is produced. The approach of reusing the current scripts for testing presents a challenge to the Adaptor. Current scripts trap the errors when they fail their execution and report the fail to the current testing framework. As errors are trapped, scripts seem to execute correctly always. This means that the Adaptor never observes an abnormal execution and never reports a failing test case. Table 5.2 presents the verdicts for the execution of each action in a test case as reported by the model-based tool SmartMBT and the current TET/CTI framework.

Table 5.2 shows that the SmartMBT tool generates a *pass* verdict for the test cases that create a new file system while the TET/CTI framework reports an error and produces a *fail* verdict. Results for action *format7* present the same difference between SmartMBT and

Table 5.2: Execution verdicts for test cases in SDTS suite

Test action	SmartMBT	TET/CTI
format1	pass	pass
vtoc1	pass	pass
format3	pass	pass
newfs1	pass	fail
format1	pass	pass
newfs3	pass	fail
fdisk3	pass	pass
format6	pass	pass
format7	pass	fail
newfs2	pass	fail

TET/CTI reports. The reason for these differences resides in the fact that the scripts in the SDTS trap the errors and then finalise their execution normally.

Section 5.2.4 presents a more in-depth discussion of the results of the previous modelling and test generation. However, a brief analysis of the results concludes that the implementation of a model-based framework that reuses the code and structure of the test suites as they are currently defined does not improve significantly the flexibility of the test suite. The major difference between the test cases in the current suites and the ones generated, resides in the execution order of the test cases. This normally configures different test scenarios, however, given that each test case "initialises and cleans" the state of the system before and after execution, test scenarios do not really change. Therefore, there is a need for a different modelling effort that can expand the benefits of model-based testing into the current test suites.

5.2.3 Test generation with fine-grained models

From our previous experience we know that the test purposes in the SDTS suite are not very flexible because of their high level of abstraction. Intuitively, providing more fine-grained control over the operations executed by the OS increases the number of possible test scenarios. This second exercise uses models that represent fine-grained functionalities instead of current SDTS scripts. These new models concentrate in the basic functionalities of the HBA storage driver and restrict themselves to the list of functions described in Section 5.2.1 and to the

elements that take part in those functions.

5.2.3.1 Modelling the HBA driver

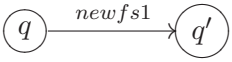
Figure 5.7 shows an example of the current modelling approach and the differences that it presents with respect to the previous one.

- The first part (Part 5.7(a)) of this figure shows the test purpose *newfs1*. This representation assumes that the system is not cleaned after execution and, thus, there is an actual transition of state.
- In Part 5.7(b), the same "newfs1" test purpose appears represented using a fine-grained list of the HBA driver functions. However, this representation still suggests that after a SMI label has been attached to the disk, a new partition has to be created. Then, in sequence, this partition is converted into a UFS file system which is finally mounted.
- Part 5.7(c) resembles more the aim of greater flexibility where different test scenarios are derived from the same model. For example, this model easily generates a test scenario where the disk is formatted and a partition is created and then deleted, or where a file system is created but never mounted because the disk is re-formatted.

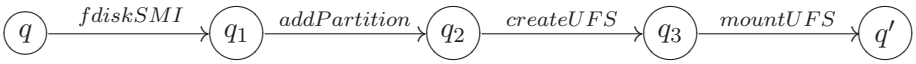
The natural question that arises is how do we model the HBA driver so that these scenarios can be derived from the model. Figure 5.7 provides hints for answering this question. This model requires a set of actions that represents closely the basic functionalities of the HBA driver. These actions drive the definition of the states. States of this model need to represent the elements over which these actions perform.

Following the description of the HBA driver in Section 5.2.1, consider as elements of the system:

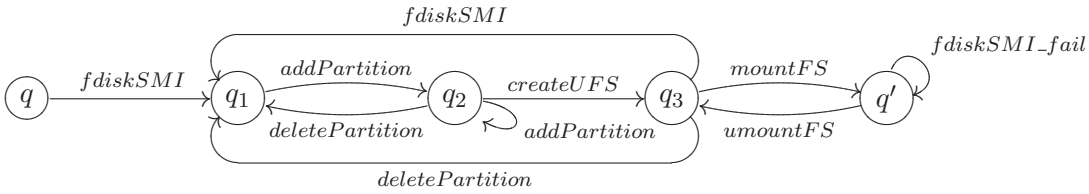
- a set of devices (disks) that can be available for the tests,
- for each device, a set of partitions, and
- a set of mount points where file systems can be mounted.



(a) New file system test purpose



(b) Sequence of actions in the "newfs1" test purpose



(c) Different scenarios derived from "newfs" test purpose

Figure 5.7: Model representing a test that creates and mounts a partition

Each device has attached the following information: name, size, formatting (either they have a SMI or an EFI label), and a boolean value that indicates if they are available for disk operations. Similarly, each partition has the following information attached: name (how it is referred to), number (each partition has an ordinal number in the partition table of the disk), size, and type of file system it contains. Finally, the mount points have a name linked to the name of the partition mounted on them.

Consider the quantity and possible values of the information contained in this model and compare it with the set of boolean functions in our previous model. The definition of states for this model is more complex that the one presented for the SDTS suite.

The states of the system are defined by all the combinations of possible values that the previous elements can have. The enumeration of states of the system before hand turns then into a difficult task. Restrict the devices and the mount points to include a single element each. Restrict also the partitions to consider only one of each type of partitions included in OpenSolaris (slices and partitions). Even with these restrictions, the count reaches approximately 144 combinations that include additionally three file systems (UFS, FAT32 and ZFS), two formatting labels, the EFI and SMI labels, and a special value labelled *none* that

Table 5.3: States in the model of the HBA driver

State	Device				Partition				Mount
	name	size	label	available	name	number	size	file system	point
s_1	c3d1	500	none	true	none	none	none	none	none
s_2	c3d1	500	SMI	true	p0	1	500	none	none
s_3	c3d1	500	EFI	true	p0	1	500	none	none
η	c3d1	500	EFI	true	p0	1	500	ZFS	none
s_4	c3d1	500	EFI	true	p0	1	500	UFS	none
η	c3d1	500	EFI	true	p0	1	500	UFS	mp1

represents the absence of a file system, either because it has not been created yet, or because a formatting label has not been applied. Table 5.3 shows a partial view of the states that are part of the model.

The first column of Table 5.3 includes a state name for each combination. However, there are some combinations of values that are not considered valid states. Such combinations are marked with the symbol η in the state name column. These "not valid" states appear because some values are dependent on others. Consider, for example, the ZFS file system, it can only be created into a device with a SMI label. Thus, the combination of a ZFS file system with an EFI label can never occur. A similar situation presents itself in the last combination. A device cannot be available for formatting, or modifying a defined partition, if this partition is mounted.

Definition of states for a fine-grained model turns complex not only because of the number of combinations of values for the different elements but also because of the analysis that is required to determine if certain combinations are valid. This is why, in general, states are not enumerated but defined abstractly. Then, different algorithms can be applied over the abstract definitions in order to generate a complete set of states if required.

The briefly stated list of functions of the HBA driver (see Section 5.2.1) provides an initial base to define the actions considered in the new model. Initially, these actions include *fdiskSMI*, *fdiskEFI*, *addPartition*, *createUFS*, *createFAT32*, *createZFS* and *mountFS*. Additionally, actions such as *deletePartition*, *umountFS*, and other actions that represent failure of the previous actions, such as *fdiskSMI_fail* and *mountFS_fail*, increase the flexibility of the model.

In general the execution of an action should fail if the state of the system does not allow it. In other words, each action defines (explicitly or implicitly) a list of states from where its execution is allowed. An attempt to execute an action from a state not defined in this list results in a failure. Consider the case of the formatting operation of a file system, the operating system formats a device only if its file systems are not mounted. A (normal) test usually verifies that condition before executing the formatting action. Our failing test, on the contrary, tries to force a scenario where the formatting is executed over mounted partitions, therefore, expecting the action to fail. Section 5.2.4 discusses how the ability to force the execution of an action in different states increases the flexibility of the test suite.

5.2.3.2 Test generation

The generation of test cases for the fine-grained model follows the same random approach as the previous high level model. However, this time the system is observed in different states and the set of available actions changes as the state of the system changes.

The selection of the random approach is justified because it allows to, eventually, execute each action in the model in a number of different states. An exploratory approach leads to the same result but it requires interaction with the tester. Other algorithmic approaches, such as Chinese Postman, cannot be applied because the states in the model depend on data that is not generated before hand but is instead produced as the generation process progresses. The generation process becomes aware of more states of the system as it executes more actions. For example, after a random execution of 200 actions the number of visited states in the test sequence is 35. An increase of the number of executed actions to 1000 turns the number of visited states into 63. As an example of a generated sequence of actions, consider the subsequence: *initialise*, *fdiskSMI(c3d1)*, *fdiskSMI(c3d1)*, *makeMountDir(mp1)*, *fdiskEFI(c3d1)*, *fdiskSMI(c3d1)*, *addPartition(c3d1,fat32)*, *formatSlice0(c3d1)*, *createUFS(c3d1,s0)*, *fdiskEFI(c3d1)*.

A graphical representation provides an intuitive idea of the differences between the tests generated in the present and the previous approaches. Consider the state chart shown in Figure 5.8. This state chart shows the initial 4 actions of the test sequence as they were generated. It also shows other possible paths in state *s3*. This means that the test sequence

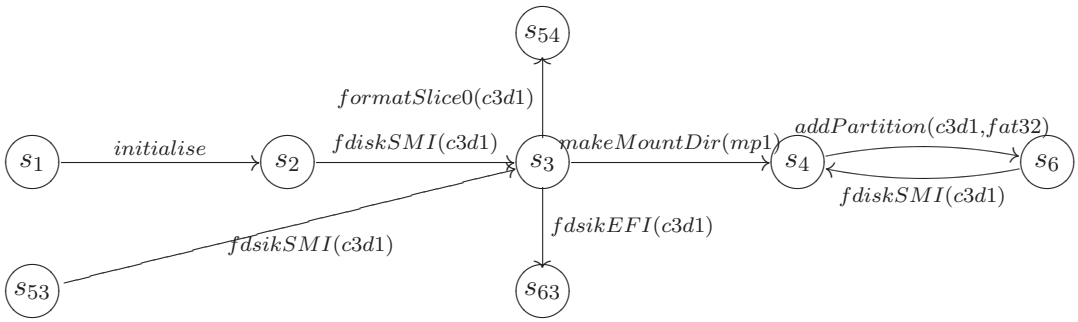


Figure 5.8: Extract of the state chart for the generated test sequence.

tests different actions under the same scenario. The transition with label *fdiskSMI(c3d1)* between states *s53* and *s3* tests the action *fdiskSMI(c3d1)* under a different scenario than the first subsequence but expects to arrive at the same state.

Section 5.2.4 discusses in more depth the advantages of the generated test sequence. The following section discusses aspects related to the execution of this test sequence.

5.2.3.3 Test execution

Just as a high level model, a fine-grained model requires an Adaptor module and a set of scripts in order to connect to the SUT and execute the tests automatically. In general, these new scripts are not very different from the ones in the SDTS suite but are instead a break up of them. To illustrate this better, Table 5.4 shows how the code of the scripts in Figures 5.3, 5.4 and 5.5 is divided among the new actions of the model. Action *fdiskSMI* takes the relevant part of function *label_SMI* that formats a disk with a SMI label, *createUFS* and *mountUFS* split the relevant code of function *build_ufs* that creates a file system on a given partition and mounts it, respectively.

The scripts in Table 5.4 do not contain the complete code from the original script. Calls to the CTI framework, *cti_report* and *cti_fail*, for example, are not present anymore. The purpose of such code in the original script was to trap errors and provide a test report in the CTI/TET framework. The new scripts replace these functions with a return code. Although this replacement is not necessary for executing the tests, it is convenient because it enables the Adaptor to determine when a script has failed and to report it to the Arbiter in the testing tool indicating that the test has *failed*. Trapping errors as in the original scripts will

Table 5.4: New model actions and their corresponding script code

fdiskSMI	`if ! fdisk -B /dev/rdsk/\$stds_disk; then`
	`  return 1`
	`fi`

addPartition	`print "partition\n0\n\n\n0\n${size}gb \`
	`  \n1\n\n\n0\n0\n2 \`
	`  \n\n\n0\n0\n3\n\n\n0\n0\n4\n\n\n0\n0\n5\n\n\n0\n0\n6\n \`
	`  \n\n0\n0\n7\n\n\n0\n0\nlabel\nq\nq" \`
	`  > ${testdir}/create_slice0.txt`
	`if ! format -f ${testdir}/create_slice0.txt \`
	`  -s $sdts_disk; then`
	`  return 1`
	`fi`

createUFS	`if ! yes | newfs $rdev >/dev/null 2>&1; then`
	`  return 1`
	`fi`

mountUFS	`if ! mount $bdev $mp >/dev/null 2>&1; then`
	`  return 1`
	`fi`

umountUFS	`if ! umount $mp  > /dev/null 2>&1; then`
	`  if ! umount -f $mp  > /dev/null 2>&1; then`
	`  return 1`
	`  fi`
	`fi`

mean to the Adaptor that the scripts always terminate correctly and thus will indicate that the test *passed* even when it produced an error report.

5.2.4 Discussion

The first exercise, that reuses completely the current scripts in the SDTS suite shows that our approach generates different scenarios not considered in the original test case definitions. However, this does not translate into a real advantage because the scripts that execute the test cases erase any changes in the system state. In other words, it does not make any difference to execute the test purposes in different orders because, before and after execution, the state is always returned to a predefined state.

The modelling task proved to be of value because it revealed an error in the function *less1Tb*. A faulty condition in the code led this function to report the wrong state of the system. Although different testing approaches other than the model-based approach are

probably able to reveal the same error, it is correct to claim that the effort allocated to produce the models led, or at least contributed, to revealing this error.

The second exercise provides the test suite with increased flexibility. This exercise not only increases the number of test scenarios but also diversifies the states in which the system is subject to test. The addition of failing actions also increases the ability of the tests to verify that certain operations are not allowed in predefined states (e.g., a user cannot format a disk that has a partition mounted). Therefore, it gives more fine-grained control over the system operations.

One of the main goals of this exercise is to provide the test suite with the ability to execute different test scenarios. The number of scenarios is the product of different executions for the operations in a given test purpose. Two operation executions are different if they are performed in different states. Consider the first test purpose of the *newfs* test case (shown in Figure 5.7(b)). In the original test purpose there is only one test scenario, mainly because it executes each operation in a unique state. The model-based approach, after one hundred executions of randomly selected actions gets the following count

Operation	Count	Fail operation count
fdiskSMI	3	1
addPartition	2	1
createUFS	2	-
mountUFS	2	-

In the previous list, the fail operation count shows how many different executions of *fdiskSMI_fail* and *addPartition_fail* are reported. Actions *createUFS* and *mountUFS* do not have corresponding "fail" actions in the model. Without considering the fail operations, the number of test scenarios for this test purpose is 24. However, the fail operations execute the same operations with the difference that the expected result for them is to fail. Moreover, they can reveal bugs that the other operations can not. Considering the fail operations, the number of test scenarios increases to 48.

In summary, a model-based approach that reuses the current scripts of an existing test

suite provides more flexibility to the test suite in the sense that a reordering in the execution of test purposes leads to more and different test scenarios. However, the utility of a bigger number of test scenarios depends greatly on the way the scripts and the framework are written. In this model-based approach, also depending on the architecture of the framework and the scripts, the report of errors can present inconsistencies.

A solution that provides even more flexibility (bigger number of test scenarios and finer-control over the operations executed) and avoids inconsistencies is to include more details into the models and rewrite the scripts into more fine-grained versions. However, there is a trade-off between the flexibility and the complexity of the models and this deserves consideration. While the task of modelling current scripts can be straightforward, the modelling of a system in a more fine-grained style can demand more effort.

5.3 Vulnerability testing

Classical examples of vulnerabilities are present in web-based systems. The use of database engines to perform user's authentication, for example, leads sometimes to SQL injection vulnerabilities. Users' desire for dynamic content sometimes also leads to other injection vulnerabilities such as cross-site scripting.

The present case study aims to exemplify the use of a model-based approach in systematically testing web-based applications to reveal the presence of (security) vulnerabilities. In a first exercise, this case study focuses on two particular aspects:

- how a model can represent, in a general form, attacks engineered to exploit known vulnerabilities, and
- how models are used to identify (successful) instances of attacks against modelled applications.

This first exercise executes over a login functionality (login page) implemented using a standard web page and a database (SQL engine) which actually performs the authentication.

In a second exercise, this case study focuses on how a model-based approach, systematically and in an automated fashion, carries on the testing of a web-based application using

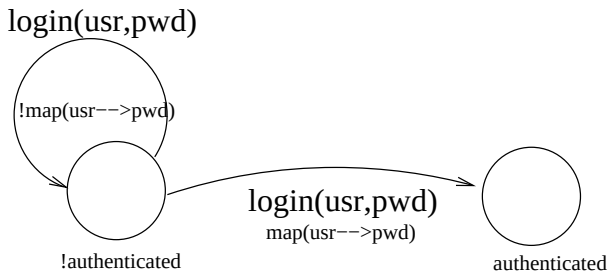


Figure 5.9: Behavioural model of a login functionality

previously identified attacks. In other words, this exercise focuses more on the execution of the test cases than on the generation of attacks. The execution of test cases still requires the generation of suitable sequences of actions that lead the system to a vulnerable state. The test sequences considered in this exercise are executed against a web application that contains pre-seeded vulnerabilities, called WebGoat[‡].

5.3.1 A web based login function

Consider the login functionality of a web application modelled as in Figure 5.9. Assume that there exists a set of user names and a bag of passwords. Assume that there exists also a boolean function *map(usr→pwd)* that returns *true* if and only if there is a relation between the user name and the password, and returns *false* otherwise. Abstractly this *map* is a total function.

The specification shown in Figure 5.9 constitutes an abstract model for the login functionality. This specification is abstract because it does not define how the map is implemented in the real system. It is true that the concept of maps has been implemented in most (if not all) programming languages and due to this, maps can be created and maintained in memory. However, it is also true that for various reasons, including persistence, performance and security, the most popular implementation of login functionalities uses a database engine in the back for verifying this mapping and the (web based) application communicates with the database using SQL queries.

A common implementation of a login functionality can be modelled as shown in Fig-

[‡]http://www.owasp.org/index.php/Category:OWASP_WebGoat_Project

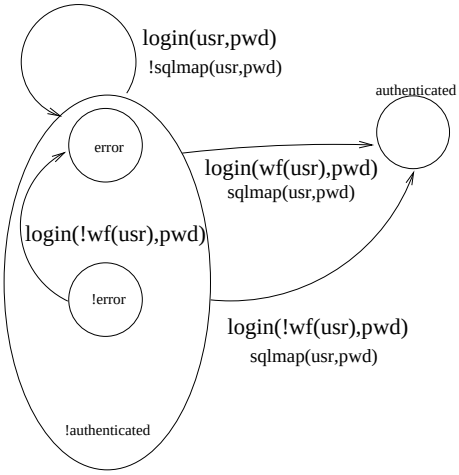


Figure 5.10: Specification of the implemented application

ure 5.10. This implementation model has two extra transitions when compared with the behavioural model. This is because the implementation model considers explicit transitions that take *ill-formed* input parameters. The concept of ill-formed (and well-formed) input parameters is introduced by the function `sqlmap`. The function `sqlmap` corresponds in the implementation to the map function used in the behavioural model. A well-formed input parameter is a parameter that does not include elements of an SQL query or resembles the syntax of an SQL query.

In the abstract model well-formed and ill-formed input are handled in the same way and the result of the login function only depends on the existence of the mapping $usr \rightarrow pwd$. However, in the implementation some ill-formed input will lead to an unspecified behaviour of the SQL engine which in turn affects the result of the login function.

Another difference in comparison to the specification model is that the *!authenticated* state presents two sub-states, *error* and *!error*. The *map* function in the specification model always returns either a true or a false value. The *sqlmap* function with some specific ill-formed input will return neither true nor false values but an error message. This behaviour is introduced by the use of the SQL engine and is captured by the *error* state.

The model in Figure 5.10 is permissive in that it models an oversight by the programmer in not taking care of the minimum recommended countermeasures against SQL injection

attacks.

With the aim of testing a system for the presence of vulnerabilities in it, the behavioural and implementation models of the system complement themselves with a model of the attacker. In general, the model of the attacker identifies specifically a vulnerable transition in the implementation model and the way it can be exploited. For the attacker, the transition `login(!wf(usr),pwd)` in Figure 5.10, represents a set of data dependent transitions. That is, the transition will end in an unsuccessful login attempt with some particular data parameters, while it will end in a successful attempt with different data parameters. The model of the the attacker shows which data will be used for this transition to end in a successful attack.

For practical purposes, models are specified using a pre- and post-condition approach. They are represented using a declarative language [98] based on the Object Constraint Language (OCL)[126]. This exercise shows two attack models written in this language, one where the attacker aims to perform an unauthorised authentication, and a second one that aims to force the system to disclose information about its structure.

The first attack, specified in Figure 5.11, defines an attack over the login operation. The pre-condition defines the data to be used for the `usr` parameter. The post-condition specifies that as a result of the login execution the system reaches the authenticated state.

```
operation: login

pre: usr ≡
     [AZaz] ∗ (′) OR (\%20)∗
     (1 = 1)|(′string′ =′ string′)(−−)
      pwd ≡
     [AZaz] ∗ (′) OR (\%20)∗
     (1 = 1)|(′string′ =′ string′)(−−)

post: !map(usr → pwd) && authenticated == true
```

Figure 5.11: An *authentication* attack

The second attack, shown in Figure 5.12, shows an attacker whose intention is to force the SQL engine running in the background to get into an error state. The SQL engine then will return an error statement containing structural data that should not be disclosed. This attack defines the use of the login operation with specific data for the `usr` parameter. The

post-condition defines that the attacker learns the column name in which the `usr` data is stored in the database.

```
operation: login

pre: usr ≡ ([AZaz]∗)(′)
     pwd ≡ ([AZaz]∗)(′) |
       ≡ (′) or (1 = 1) order by 5(−−)|
post: error && columnname ∈ error_message &&
    update_knowledge(columnname)
```

Figure 5.12: An *information-disclosure* attack

Consider that the pictorial representations in Figures 5.9 and 5.10 are written in the same declarative language as the attacker model. With all the models defined in a uniform way, proceed to generate test cases that will identify the presence of the vulnerable transitions in a real implementation.

5.3.1.1 Test case generation

The generation of test cases is based in the combination of the three models, behavioural, implementation and attacker models. Recall the definition of the *faulty contexts* presented in Section 4.1. The models usually configure one of the three contexts described in that section. The faulty contexts define, in their turn, how the models are combined for generating the tests. For example, the case of the authentication attack configures the context in Figure 4.1(a) in Section 4.1.

In the case of the authentication attack, the test generation process combines the models into the predicate $Pre^S \wedge \neg Pre^I \wedge \neg Post^S \wedge Pre^A \wedge Post^A$. The intuition behind this process is that a modelled attack ($Pre^A \wedge Post^A$) will be successful if the implementation does not correctly handle parameters that are handled by the specification ($Pre^S \wedge \neg Pre^I$) and therefore does not behave as expected ($\neg Post^S$). A particular instance of this predicate derived from the actual models is

true ∧

¬ wf(usr) ∧

¬ ((map(usr → pwd) ∧ authenticated'=true) ∨ (¬ map(usr → pwd) ∧ authenticated'=true)) ∧

usr = "' OR (1=1) – " ∨ usr = "' OR (1=1) – " ∧

¬ map(usr → pwd) ∧ authenticated' = true

This predicate differentiates between variables referenced in the pre- and post-conditions. A variable v is denoted v' if referenced in a post-condition. Then, v' refers to the value of v after the execution of the action (e.g., *authenticated'* represents the state of the system after the login operation).

The generation of a test case (or a set of them) reduces itself to the search for suitable values for the variables in the previous predicates. The implementation of the model-based framework for the present exercise uses a constraint solver module for finding this set of values. The constraint solver, built following the approach of Constraint Handling Rules [45], is based on previous work [98]. It implements internally the functions used in the predicates, such as *map* and *wf*, and is able to reduce them to boolean values.

The attack performed in the second exercise (*information disclosure*) translates into the predicate $Pre^S \wedge Post^S \wedge Post^I \wedge Pre^A \wedge Post^A$. This predicate represents the case where the implementation maintains the behaviour of the abstract specification ($Pre^S \wedge Post^S$). That is, the input parameter $!wf(usr)$ does not provide authentication. Thus, the implementation is a correct refinement of the specification. However, the implementation presents a new sub-state ($Post^I$), an *error state*, that is used by the attacker as specified in the attack model ($Pre^A \wedge Post^A$). This case configures the context depicted by Figure 4.1(b) in Section 4.1. A particular instance of this predicate combining concrete values for the context predicate is

Table 5.5: Attacks that result in unauthorised authentication

usr	pwd	authenticated
“’ or 1=1 –”	“admin”	true
“’%20or%201=1%20–”	“tester”	true
“’ or ’x’=’x’ –”	“’ or ’x’=’x’ –”	true
“admin”	“’ or 1=1 –”	true
“tester”	“’ or 1=1 –”	true
“tester”	“’ or ’x’=’x’ –”	true

true ∧

(map(usr → pwd) ∧ authenticated = true) ∪ (¬ map(usr → pwd) ∧ authenticated = false) ∧

sql_map(usr → pwd) = “error” ∧

error’ = true ∧ authenticated = false ∧

usr = “ user’ ” ∨ usr = “’ OR (1=1) order by – ” ∧

error’ = true ∧ update_knowledge(columnname)

Similarly to the first one, this exercise uses a suitable constraint solver that generates values for the variables in the predicate. These values are used as input parameters to configure a successful attack in a vulnerable application. In this case, the constraint solver includes an implementation of the *sql_map* function. Modelling the responses of an SQL server is not practical. Instead, the function *sql_map* queries a real SQL server and returns suitable responses to the constraint solver.

5.3.1.2 Results

The constraint solver used to implement the model-based approach for this case study generates 21 test cases for the authentication attack. Table 5.5 shows an snapshot of these test cases. Note that the generated test cases use different ways of forcing the *where clause* of an SQL query to be evaluated to true. As a result, the vulnerable system grants authentication to any user that provides the generated values as input parameters. The requirement of an existing valid mapping between the user and password variables in the database is over sighted by the application.

In the case of the information disclosure attack, 12 test cases are generated and a subset

Table 5.6: Attacks that force an SQL engine error

usr	pwd	authenticated	error
“user’ ”	“user’ ”	false	Unexpected token: USER in statement [user]
“user ”	“user’ ”	false	Unexpected end of command in statement [select . . .]
“admin”	“’ or 1=1 order by 5 –”	false	Cannot be in ORDER BY clause in statement [select . . .]

of them is shown in Table 5.6. The tests in this group conform to the specification in that they do not provide authentication to (usr, pwd) pairs that are not related in the database. However, they try different ways of forcing the SQL engine to fail and return an error message containing useful information that will enhance the knowledge of the attacker and will be used in future attacks. For example, the second and third attacks return the SQL query performed by the web application. This query contains at least the name of the table where users and passwords are stored and probably the name of the fields.

Note that in both cases the number of generated test cases depends on the test data provided by the data generation model. In this exercise the attacker’s model contains a suitable repository of attacks. This repository collects different attacks described in other vulnerability databases such as the OWASP repository [97]. The constraint solver uses this repository as a domain for the input parameters. Thus, in this particular case, the data generation process is carried out as a lookup into the repository. Nevertheless, different implementations of a data generator module provide different test data.

5.3.2 The WebGoat application

WebGoat is a deliberately insecure J2EE web application maintained by OWASP[§] designed to teach web application security lessons. A user can navigate through various lessons which present information on how different vulnerabilities can be exploited and avoided. There are also lab activities which are small modules that contain several kinds of real vulnerabilities such as SQL scripting and Cross-site scripting. Although it is primarily designed to

[§]Open Web Application Security Project

be a teaching environment, WebGoat is also useful to show how a model-based approach systematically automates the search for known vulnerabilities and also shows how to exploit them.

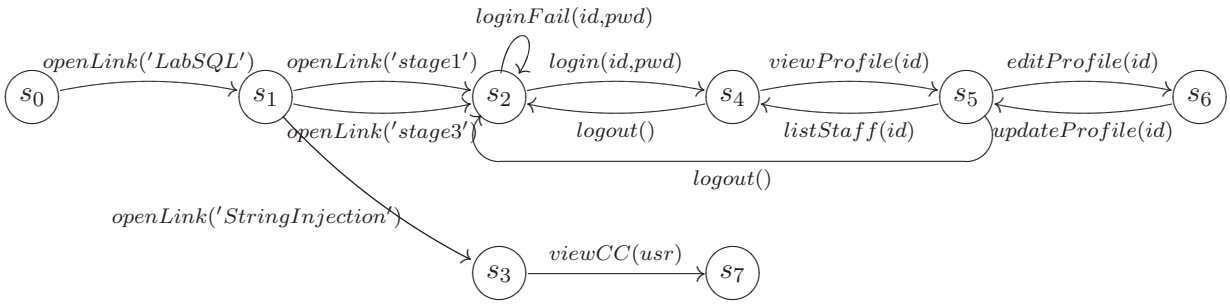


Figure 5.13: Behavioural model for the WebGoat application.

Figure 5.13 shows an extract of the behavioural model for the WebGoat application. This model shows the actual navigation that can be performed through the web interface of the application. It also shows some actions that are performed as "lab activities" such as logging into the web system of a company. For didactical purposes, the `login` functionality of the system has been divided into two actions, *login* and *loginFail*. Although both are triggered by the same mechanism, a button on the login web page, the *loginFail* action represents the case where the input parameters *id* and *pwd* are incorrect and thus the user is not logged in.

Figure 5.14 shows the attacker model used in this exercise. This model represents the assumptions and knowledge of a malicious user. In this case, the attacker knows (or assumes) that the implementation is vulnerable in state s_2. This vulnerability allows the attacker to

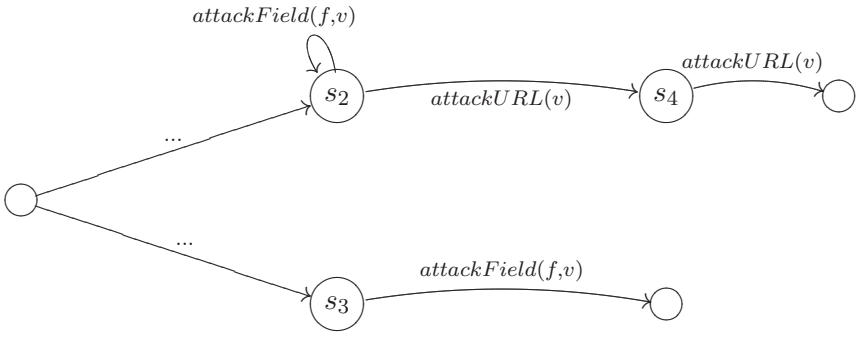


Figure 5.14: Attacker model for the WebGoat application. The attacker believes (or knows) that he can replace the *login(employee,password)* function in the behavioural model by an attack over the URL in the bar address (*attackURL(v)*).

login using a particular set of input parameters that otherwise will make the login attempt fail. The model represents this with the action *attackField*. The attacker also knows that if the input is sanitised in the field box of the web page, the values can still be injected in the URL via the address bar. This is represented in the model by action *attackURL*. In both cases, in state s_2, *attackField* and *attackURL* represent the execution of the *loginFail* action.

The attacker model also represents the assumption that in state s_4, although there is no field to be attacked, the *http request* generated by the *viewProfile* action can be tampered with in the address bar to perform an attack. This is represented by action attackURL (with a suitable attack string). Finally, the attacker model also shows that in state s_3 the web application presents a field that can be attacked.

As part of the attacker's model, the attacker also knows which values for the parameters *employee* and/or *password* will more likely lead the attack to success. Then, the attacker defines a repository of values for these parameters so that the right values (for SQL injection attacks) can be retrieved from it.

5.3.2.1 Test generation and execution

For this particular case, the implementation model is represented by the WebGoat application itself. The main function of the implementation model is to show if and when there are vulnerable states that can be exploited. The WebGoat application contains vulnerable states and explicitly defines which they are. Moreover, by using an interactive (on-line) approach, the implemented testing framework generates one test case from the combination of the behavioural and attacker models, and then verifies if this test succeeds against the real implementation. This is done several times as needed and only tests that succeed against the real implementation are considered to be attacks.

This exercise uses the SmartMBT as the implementation of the model-based framework presented on this thesis. Then, for this exercise the models are directly written as first order logic structures instead of the OCL-based language. Additionally, the Prolog engine implements the constraint solver used to identify suitable attack transitions. In Figure 5.15 this work presents an extract of the Prolog code that defines the model of the attacker for

```
params(attackURL(AttackString)):-
member(AttackString,
['&amp;action=login&amp;employee_id=112&amp;password=x\' or \'1\'=\'1',
 '&amp;action=login&amp;employee_id=101&amp;password=x\' or \'1\'=\'1'],
 '&amp;action=viewProfile&amp;employee_id=101 or 1=1 ORDER BY salary DESC']).

transition(attackURL(AttackString)) :-
getv(currentpage,Currentpage),
getv(strSQLCompleted,StrSQLCompleted),
getv(result,Result),

(Currentpage = 'stage1',
_Currentpage = 'loggedin'),

_Result = '',

setv(strSQLCompleted,_StrSQLCompleted),
setv(currentpage,_Currentpage),
setv(result,_Result).

?-init_method_weight(attackURL,1).
```

Figure 5.15: AttackURL action of the attacker's model

the SmartMBT tool.

In the code shown in Figure 5.15, the section `params` defines the strings that can be used as input parameters for the *attackURL* action. Inside the `transition` section this code defines that this attack will be performed when the WebGoat application is in the page labelled as 'stage1' and that as a result the attacker expects to reach the 'loggedin' page.

The test generation (an execution) process in the framework uses the interactive approach of SmartMBT to generate a test sequence. Both models, behavioural and attacker models, are translated to the Prolog language and provided as input to the SmartMBT tool. An additional action *initialise* is included into the Prolog code so that it brings the models to their initial state. The generated sequence contains actions that perform different SQL injection attacks over the WebGoat application. The test generation is summarised as follows:

- Execute action *initialise* to start the test;
- Execute actions *openLink('Lab SQL')* and *openLink('stage1')* from the behavioural

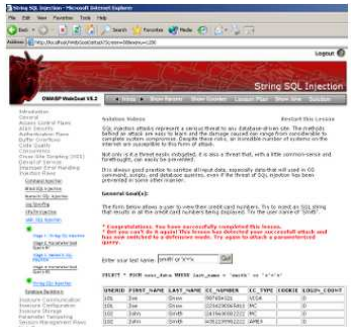
model;

- given that we are in the "vulnerable" state s_1, execute action *attackURL* with the second string shown in the Prolog code as parameter. This execution succeeds authenticating the attacker as the admin user (employee with Id 101);
- follow the transition *logout* in the behavioural model;
- open a different activity by executing action *openLink('stage3')*, which leads to state s_1;
- login as an employee different from the administrator (repeat the previous attack with a different employee id);
- in the new vulnerable state s_3 switch to the attack model and execute the action *attackURL* (with a suitable parameter) in replacement of *viewprofile* which leads to disclose the profile of the first employee in the database's table;
- open a third activity executing action *openLink('String Injection')*;
- in state s_2 execute the action *attackField* with the string *smith' or 'x'='x'*; and
- the list of credit card numbers for all employees (users) is disclosed.

The generated sequence contains three attacks which have been executed successfully against the WebGoat application. Each action is executed by a script in the Adaptor module of the tool. These scripts are written in Watij (a Java port of the Watir - Web applications testing in Ruby) because they need to control the web browser in an automated fashion. The implementation of the Adaptor module also uses the JUnit framework to manage the execution of the test cases. Figure 5.16 shows the result of the execution of the last step in the test sequence, a successful disclosure of credit card numbers.

5.3.3 Discussion

One of the main advantages of using models to generate tests for security vulnerabilities in web based systems is that the models allow one to specify precisely all conditions and



General Goal(s):

The form below allows a user to view their credit card numbers. Try to inject an SQL string that results in all the credit card numbers being displayed. Try the user name of 'Smith'.

*** Congratulations. You have successfully completed this lesson.**
*** Bet you can't do it again! This lesson has detected your successfull attack and has now switched to a defensive mode. Try again to attack a parameterized query.**

Enter your last name: smith' or 'x'='x Go!

```
SELECT * FROM user_data WHERE last_name = 'smith' or 'x'='x'
```

USERID	FIRST_NAME	LAST_NAME	CC_NUMBER	CC_TYPE	COOKIE	LOGIN_COUNT
101	Joe	Snow	987654321	VISA		0
101	Joe	Snow	2234200065411	MC		0
102	John	Smith	2435600002222	MC		0
102	John	Smith	4352209902222	AMEX		0
103	Jane	Plane	123456789	MC		0
103	Jane	Plane	333498703333	AMEX		0
10312	Jolly	Hershey	176896789	MC		0
10312	Jolly	Hershey	333300003333	AMEX		0
10323	Grumpy	White	673834489	MC		0
10323	Grumpy	White	33413003333	AMEX		0
15603	Peter	Sand	123609789	MC		0

Figure 5.16: Credit card numbers disclosed by an SQL injection attack

assumptions under which the testing is performed. Implementation models represent the specific characteristics that make an application vulnerable. Attacker models not only specify the data parameters that trigger the exploitation of an existent vulnerability but also the system interface (usually an action or operation) used to perform the exploit. This provides finer control over the test generation in the sense that for different operations, different sets of data parameters can be specified and linked to different sets of variables in the application. Ultimately, test cases generated from these models will reveal vulnerabilities that present the modelled characteristics and that are exploited as specified, and will not provide any guarantee otherwise.

The present study included cases of multiple step attacks. That is, attacks that were performed over two or more actions in a predefined order. Automated scripts can execute such kind of attacks. However, the use of models provides additional flexibility. A sequence of actions composing an attack can be included as a subset of a longer generated sequence of actions provided that the first sequence triggers the same transitions as specified into the attack model. This characteristic of model-driven test generation leads to disguise attack signatures. Disguised attacks can reach the applications even when (other) network level defences have been put in place.

Although not explicitly reported in the description of this case study, the experience obtained by performing it suggests that the use of models in this testing context reduces the time of test execution. Executing tests against a web based SUT requires time to connect to the web application, request the execution of the test and interpret the results. When a GUI (a web browser rendering the web pages) is used to access the application's functionalities, the execution time is noticeable. Thus, executing attacks that can be flagged a priori as unsuccessful wastes execution time. Implementation models allow the tester to filter a set of defined attacks. These models identify those attacks that will be successful under the conditions and assumptions specified by the models, thus reducing the time and effort of executing tests already known as being unsuccessful.

In summary, there exist different sources that define known vulnerabilities and how to exploit (attack) them. These vulnerabilities and their associated attacks can be represented

as LTS models. General attack models can be derived from vulnerability repositories and can be made more specific, thus defining which operations (actions) they are applied against in a modelled application. The way models are combined into the present testing framework reduces the time for the execution of a test suite if there exist test cases that will not lead to successful attacks. Finally, model driven test generation can increase the probability of an attack being successful even if network level defences are in place.

5.4 Privacy testing

The present case study focuses on showing how a model-based approach can be applied to test and verify that a given implementation does comply with a predefined privacy policy. This study is composed of two exercises. The first exercise focuses on web browsing privacy, more specifically, it models a policy known of as the *same origin* policy which defines how the privacy of web browsers' users should be protected. In the second exercise, this study focuses on the handling of conflicting rules in the privacy policy and the handling of obligations in a particular rule. This second exercise focuses on the modelling and the generation processes.

5.4.1 Web browsing privacy: the same origin policy

The same origin policy is a privacy principle included in most web browsers that allows cookies and Javascript from different sites to coexist without disturbing each other. Jackson et al. [66] show that this principle had not been applied to other features like the caching and the history features and that the lack of its implementation enabled tracking the user behaviour without consent.

In the case of the caching feature, Jackson et al. [66] consider that there is a site A to which a user provides information. Then, there is a malicious site B that intends to access such information. They suggest that B can gain access to the information by embedding site A. In other words, B creates a framed page and leads the user to open A inside one of the frames. The host of site B becomes the *top level domain* because it "owns" the framed page. If site A was cached before, the web browser will use the cached information to display the

content. Then, site B gains access to the cached information. Thus, the same-origin policy for this feature states that "if the same site embeds a previously cached content, it is appropriate to allow the existing cached content to be used". It also states "if a different site embeds the same content, the existing cached content may not be used".

In the case of the history feature, Jackson et al. consider that there is a malicious site A that intends to know which sites a user has visited before. With this aim, site A includes links to other sites, such as site B. When site A is displayed, it observes features of its links such as the font colour. Then, if the link to site B has a different colour, A knows that B has been visited before. In summary, site A forces the web browser to use the history feature by adding links to other sites. For this scenario, the same origin policy describes two conditions, that is, "a hyper link located on page A and pointing at a visited page B would appear unvisited unless both of the following conditions are met:

- The site of page A is permitted to maintain a persistent state, and
- page A and page B are part of the same site, or the user has previously visited the exact URL of page B when the referrer was a page from site A."

5.4.1.1 Privacy model

The central element of the framework in the privacy testing domain is the privacy policy. However, the policy description presented before in natural language provides not much help if the intention is to generate data in an automated way. Even if the generation of data is performed manually, natural language descriptions can lead to errors. Therefore, the policy needs to be described into a more structured way like it is defined in Section 4.2.

Before describing the privacy policy in the framework's privacy modelling language, it is important to be aware of the following considerations. The same origin policy considered in this case study is linked to the use of caching and history features of the web browser. Assume actions *getCached* and *markVisited* that are executed by the web browser when it uses its cache and history respectively. The rules in the privacy policy refer then to the execution of these actions. These actions are executed over a single data category, called *URLCat*.

Data elements in the *URLCat* category are URLs. A URL designates the address of a web page on the internet and gives access to the contents of such a web page. Any web page address contains a *host* and a *path*, e.g., *mysite.com/pubs/mydoc.html* is an address, the *host* is *mysite.com* and the *path* is *pubs/mydoc.html*. Consider the function *host(URL)* that returns the host of a given URL.

The user categories in the privacy rule define who can (or cannot) get access to a data element using the action referred to by the rule. In this study, there are two user categories. Consider *Embed-host* a category that represents the top-level domain of a framed web page that is currently displayed by the web browser. Consider also *Ref-host* a category that represents the host of a page that contains links to other pages.

Finally, consider that the web browser has a repository *cache* that stores cached URLs and their top level domain (TLD), and a repository *history* that stores previously visited URLs and the URL that referred to them (REF). Assume functions *inCache(*URL,TLD*)* and *inHistory(*URL,REF*)* that return a Boolean value *true* if the respective record in the repository exists, and otherwise return *false*.

Consider the same origin policy defined in Table 5.7. This description is more structured than a natural language representation and follows the definitions presented in Section 4.2.

Table 5.7: Rules for the same-origin policy. Rule 1 refers to the caching feature. Rule 2 refers to the history feature.

rule	ρ	a	d	ag	c	O β	δ	υ
1	deny	getCached	URLCat	Embed-host	*inCache*(URL,TLD) ∧ TLD≠ nil ∧ (embed-host ≠ TLD)	-	-	-
2	deny	markVisited	URLCat	Ref-host	inHistory(URL,REF) ∧ ref-host ≠ host(URL) ∧ ref-host ≠ host(REF)	-	-	-

5.4.1.2 Behavioural model

The same origin policy refers to privacy issues in the use of a web browser. Consider the behavioural model of a web browser depicted in Figure 5.17. Note that to reduce the com-

plexity of the graphical representation of the model, it uses *actions* with *variables* as parameters. Given a set of values X known of as the domain of a variable *var*, action *a(var)* is an abbreviation for all actions *a(x)* such that x is a value in X. In this case study x is an instance of *var* and, extensively, *a(x)* is an instance of *a(var)*.

The model in Figure 5.17 shows that the main behaviour of a web browser is to visit different websites. It also shows that (at least) three ways of visiting (or opening) a website are available to users. These three options are represented by actions *open(*URL*)*, *link(*URL,REF*)* and *frame(*URL,TLD,REF*)*. Action *open* refers to the opening of a web page by writing its address in the address bar. Action *link* refers to the opening of a page in the top window by clicking on a link. Finally, action *frame* refers to the opening of a page inside a frame of the current page by clicking a link. For all three actions, instances of the variable URL designate the address of a web page. When executing action *link*, instances of URL refer to the address of the linked web page while instances of REF refer to the address of the current web page. In the case of action *frame*, instances of URL and REF have the same role as in action *link* and instances of TLD refer to the host of the current framed page, the current top-level domain.

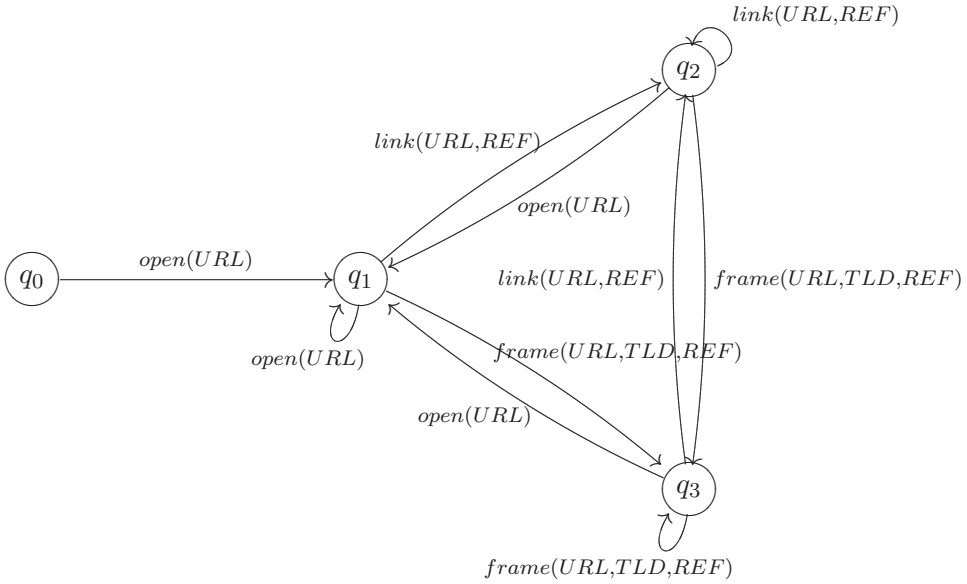
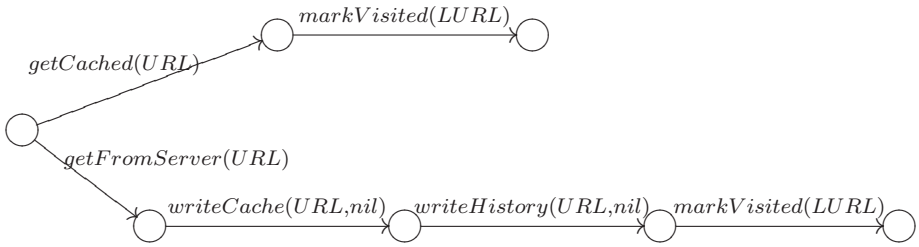


Figure 5.17: Behavioural model of a web browser

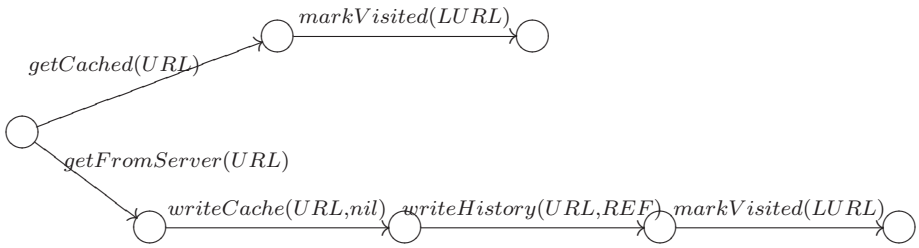
A web browser that implements these operations without using a cache, this is, that always communicates with an external server in order to get a web page, will always comply with the

same origin policy. In the same way, a web browser that does not include the history feature, and thus does not show links of already visited pages in different colours or fonts, will never have problems with the same origin policy. However, as they are desirable features for users, they are widely implemented.

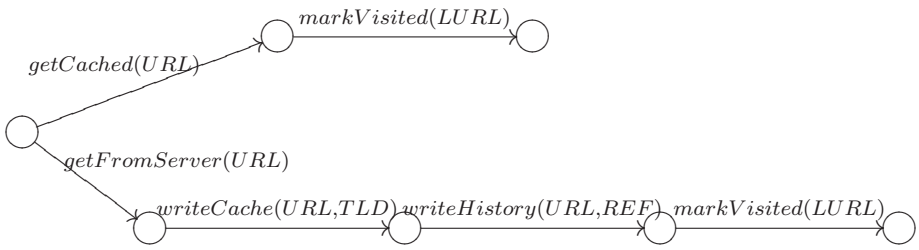
The caching and history features of a web browser are modelled as two repositories *cache* and *history* respectively. The *open*, *link* and *frame* actions use and modify these repositories in slightly different ways. To explain how these actions work internally, consider the state transitions abstractly depicted in Figure 5.18.



(a) Internal workings of open(URL) function



(b) Internal workings of link(URL,REF) function



(c) Internal workings of frame(URL,TLD,REF) function

Figure 5.18: Implementation details of caching and history features

For all three actions, whenever a user tries to open a web page, the browser gets it from the

cache repository if possible, otherwise, it gets it from the server and updates the repositories. A web page may contain *links* to other pages. The *markVisited(*URL*)* action "marks" a link to URL as visited if it appears into the *history* repository.

When updating repositories, action *open* inserts a record (URL,nil) into the cache repository. The special value `nil` indicates that the contents of URL are cached without being linked to any top-level domain. Similarly, *open* inserts the record (URL,nil) in the history, meaning that URL is visited without any referral. In the case of the *link* function, the record (URL,nil) is inserted in the cache, but a record (URL,REF) is inserted into the history, meaning that URL is being visited from a link contained in REF. Finally, in the case of the *frame* function a record (URL,TLD) is inserted into the cache meaning that the content of URL is being cached under the domain of TLD. A record (URL,REF) is inserted into the history with the same meaning as in *link*.

5.4.1.3 Linking privacy and behavioural models

Rules on a privacy policy complement the behavioural model in order to describe privacy properties of the system. These rules refer to the execution of actions in the system's model. However, notice that the action *getCached(*URL*)* that appears in *rule 1* of Table 5.7 does not appear explicitly in the model (Figure 5.17). Nevertheless, this action is executed when actions *open*, *frame* and *link* are executed and the browser has the required content in its cache. For practical purposes, these actions replace *getCached* in the privacy rule generating three rules as shown in Table 5.8.

Similarly to the case of *rule 1*, *rule 2* of Table 5.7 refers to an action that is not explicitly defined in the model. However, action *markVisited* is executed whenever actions *open*, *link* and *frame* are executed. Therefore, *rule 2* translates into the three rules shown in Table 5.9.

These new privacy rules are linked explicitly to the actions defined in the behavioural model. Hereafter this study refers mainly to these privacy rules but always still considering the rule that originated them. These rules include conditions expressed as logical predicates over variables. To be able to evaluate such conditions, values must be assigned to these variables. The next section discusses how data is generated for these variables.

Table 5.8: Rules for the caching feature in the same origin policy.

rule	ρ	a	d	ag	c	O β	δ	υ
r_1	deny	open	URLCat	Embed-host	*inCache*(URL,TLD) ∧ TLD≠ nil ∧ (embed-host ≠ TLD)	-	-	-
r_2	deny	link	URLCat	Embed-host	*inCache*(URL,TLD) ∧ TLD≠ nil ∧ (embed-host ≠ TLD)	-	-	-
r_3	deny	frame	URLCat	Embed-host	*inCache*(URL,TLD) ∧ TLD≠ nil ∧ (embed-host ≠ TLD)	-	-	-

Table 5.9: Rules for the history feature in the same origin policy.

rule	ρ	a	d	ag	c	O β	δ	υ
r_4	deny	open	URLCat	Ref-host	*inHistory*(URL,REF) ∧ ref-host ≠ host(URL) ∧ ref-host ≠ host(REF)	-	-	-
r_5	deny	link	URLCat	Ref-host	*inHistory*(URL,REF) ∧ ref-host ≠ host(URL) ∧ ref-host ≠ host(REF)	-	-	-
r_6	deny	frame	URLCat	Ref-host	*inHistory*(URL,REF) ∧ ref-host ≠ host(URL) ∧ ref-host ≠ host(REF)	-	-	-

5.4.1.4 Data generation model

The previous sections describe how a web browser works in general and how it is modelled. That description uses variables to refer to the URLs of the web pages opened by the web browser, and to their *hosts*. However, without concrete values for these variables the model or any derived behaviour is not able to show compliance to the policy. Thus, the test generation process needs to define a domain of values for the URL variables. This is not as easy as it seems. The web page referred to by a URL can link to others in different ways, i.e., forcing the browser to use the *link* or the *frame* operations to open them. Then, the domain definition needs to describe not only the URLs but also how these URLs are linked.

Consider the scenario depicted in Figure 5.19. In this scenario the variable URL is replaced by the following instances:

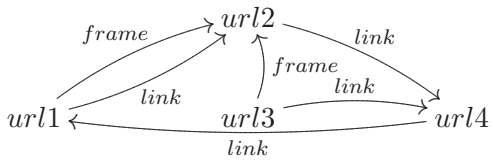


Figure 5.19: URL instances and their links

- one.pps/url1.html – **(url1)**
- two.pps/url2.php – **(url2)**
- three.pps/url3.html – **(url3)**
- four.pps/url4.php – **(url4)**

the short names (in bold) for each instance hereafter replace the complete URL to enhance readability.

Figure 5.19 shows how the URL instances are related one to another. An edge between two urls (*url* and *url′*) represents an *html link* in *url* pointing to *url′*. The label of the edge indicates which action of the behavioural model is used to open *url′*.

The defined URL instances also replace the variable REF when needed. Additionally, instances of *host(*URL*)* replace the variable TLD when it is required.

In Figure 5.20, this study shows a suitable set of grammars used in the data generation process. Action *frame*, for example, can be executed only with "url2" as its URL parameter. This reflects the fact that the pages that contain frames have links only to the "url2" page (see Figure 5.19). Similarly, every page has at least one link to open another page in the top window. However, if the current page *cp* is "url1" this link points to "url2" while if *cp* is "url2" this link points to "url4".

In the grammar corresponding to the *open* action, the guard predicates use the function *inHistory()* with the intention of forcing the test generation process to exercise the links included in different web pages. For example, *inHistory(X,'two.pps/url2.php')* returns true if there is a page X that was visited using a link from "url2". If there is not such a page, then the grammar generates "url2" as the value for *open*. This in its turn enables the *link* link in "url2". If the page X exists, then at least one link in "url2" has been used and the grammar

does not generate this value, enhancing the probability of another value being generated.

Note than in the grammars, an empty guard condition `[ ]` is always evaluated to **true** and the notation `[;]` is translated to *no operation*.

5.4.1.5 Test generation

In the previously described scenario with the defined concrete values for the variables, the test generation process produces a set of test cases by applying the algorithm presented in Section 4.2.2. Note that in the following, whenever the notation *action(var)* appears, it actually refers to an instance of the action. Extensively, a reference to state q refers to an instance q^i of q.

The first step is to select the privacy rule to be tested. Select first the rules that test the caching feature, rules r_1, r_2 and r_3 from the policy in Table 5.8. Then calculate all exceptions to these rules by using the difference operation over rules defined in Section 4.2.1.2. The resulting rules are exactly the same rules r_1, r_2 and r_3. None of them is in conflict with each other because they refer to different actions.

Then, for each rule r_i, the disjoint partitions in the *condition* c define the number of test cases required to test each rule. In this case, only one test case is needed. Then, for each one of the rules in Table 5.8, that is, for each action a_i in the model, the test generation process identifies the states q for which $q \xrightarrow{a_i} q'$ holds. For example, action *open* is available in states q_1, q_2 and q_3. The same applies to actions *link* and *frame*. Therefore, each action requires to be tested in three different states. Notice that the predicate TLD $\neq$ *nil* in the *condition* guides the string selection by requiring it to have at least one *frame* action into it.

Consider the strings in Table 5.10 produced by the generation algorithm. For each action considered in the rules and for each state (in column q_F) in which this action is available there is one string that reaches this state. Strings w_6 and w_9 are exceptions because, in the current scenario, there are no strings that lead to q_3 and then execute the *frame* action. Then, for all strings but w_6 and w_9, concatenate action a to the string to obtain the final string that characterises the behaviour included in the test case.

From the generated strings, there is no guarantee on which action the browser internally

[ ]	[;]	S_{open} → URL
[!inHistory(X, "one.pps/url1.html")]	[cp'="one.pps/url1.html"]	URL→ "one.pps/url1.html"
[!inHistory(X, "two.pps/url2.php")]	[cp'="two.pps/url2.php"]	URL→ "two.pps/url2.php"
[!inHistory(X, "three.pps/url3.html")]	[cp'="three.pps/url3.html"]	URL→ "three.pps/url3.html"
[!inHistory(X, "four.pps/url4.php")]	[cp'="four.pps/url4.php"]	URL→ "four.pps/url4.php"

[ ]	[;]	S_{link} → URL REF
[cp="one.pps/url1.html"]	[cp'="two.pps/url2.php"]	URL→ "two.pps/url2.php"
[cp="two.pps/url2.php"]	[cp'="four.pps/url4.php"]	URL→ "four.pps/url4.php"
[cp="three.pps/url3.html"]	[cp'="four.pps/url4.php"]	URL→ "four.pps/url4.php"
[cp="four.pps/url4.php"]	[cp'="one.pps/url1.html"]	URL→ "one.pps/url1.html"
[cp ≠ ""]	[;]	REF → cp

[ ]	[;]	S_{frame} → URL TLD REF
[cp="one.pps/url1.html"]	[cp'="two.pps/url2.php"]	URL→ "two.pps/url2.php"
[cp="three.pps/url3.html"]	[cp'="two.pps/url2.php"]	URL→ "two.pps/url2.php"
[cp ≠ "" and ctld ≠ ""]	[ctld'=cp]	TLD→ ctld
[cp ≠ ""]	[;]	REF→ cp

Figure 5.20: Data generation grammars for same origin policy testing

Table 5.10: Sequences of actions for testing the caching feature

	String	q_F	a
w_1	open(url1) frame(url2) open(url4)	q_1	open(url2)
w_2	open(url1) frame(url2) open(url4) link(url1)	q_2	open(url2)
w_3	open(url3) frame(url2)	q_3	open(url2)
w_4	open(url1) frame(url2) open(url3)	q_1	frame(url2)
w_5	open(url3) frame(url2) link(url4) link(url1)	q_2	frame(url2)
w_6		q_3	
w_7	open(url3) frame(url2) open(url1)	q_1	link(url2)
w_8	open(url3) frame(url2) link(url4) link(url1)	q_2	link(url2)
w_9		q_3	

Table 5.11: Test cases for the history rule

Test string	Oracle predicate
w_1	gotCached(q_1) = **false**
w_2	gotCached(q_1) = **false**
w_3	gotCached(q_1) = **false**
w_4	gotCached(q_3) = **false**
w_5	gotCached(q_3) = **false**
w_7	gotCached(q_2) = **false**
w_8	gotCached(q_2) = **false**

executes when displaying the content of a web page. In other words, for each action, the web browser can choose to show a cached web page (if possible) or to get the page from the server. The rules for the caching feature only forbid an action when it displays a cached page, thus the oracle has to discern between a cached page and a server page. Consider the boolean function *gotCached(q)* that returns **true** if the content of current page in the state q was retrieved from the cache, and **false** otherwise. Then, in order to determine the verdict of the test case, the oracle predicates are defined as shown in Table 5.11. This finalises the generation of test cases for the caching feature.

Now, select the rules for the history feature (rules r_4, r_5 and r_6 in Table 5.9). These rule were generated from action *markVisited*, different from action *getCached* that generated the first rules, thus they do not present any conflict with the previous rules. The difference operation between rules does not modify them. For each one of these rules, there is only one disjoint partition in its condition c. Therefore, one test case is needed for each state in which

the actions referred to by the rules are enabled. The three actions are enabled in the three states q_1, q_2 and q_3. Thus, a set of nine test cases is required to test all the combinations.

Consider the strings in Table 5.12 produced by the generation algorithm. For each action considered in the rules and for each state (in column q_F) in which this action is available there is one string that reaches this state. String w_{18} is an exception because it requires two *frame* operations executed in sequence and the current scenario does not consider that possibility. Then, concatenate action a to the string to obtain the final string that characterises the behaviour included in the test case.

Table 5.12: Sequences of actions for testing the history feature

	String	q_F	a
w_{10}	open(url1) frame(url2) open(url2)	q_1	open(url4)
w_{11}	open(url1) open(url3) frame(url2) open(url4)	q_1	link(url1)
w_{12}	open(url3) frame(url2) link(url4) open(url1)	q_1	frame(url2)
w_{13}	open(url3) open(url1) link(url2)	q_2	open(url3)
w_{14}	open(url3) frame(url2) open(url4) link(url1)	q_2	link(url2)
w_{15}	open(url1) frame(url2) open(url3) link(url4) link(url1)	q_2	frame(url2)
w_{16}	open(url3) frame(url2)	q_3	open(url3)
w_{17}	open(url1) open(url3) frame(url2)	q_3	link(url4)
w_{18}		q_3	

Similarly to the rules for the caching feature, the rules in Table 5.9 do not forbid the execution of actions *open*, *link* and *cache* in general but the execution of a particular instance that internally executes *markVisited* over a defined URL value. The oracle predicate must determine if the forbidden action *markVisited* was executed over a defined URL.

Consider the boolean function *shownVisited(*URL*)* that examines the page currently displayed by the web browser and returns **true** if the given URL is shown as visited before and **false** otherwise. The oracle is defined by the predicate *shownVisited(*URL*)* = **false** for appropriate instances of URL as shown in Table 5.13.

5.4.1.6 Test execution

In this phase the model-based framework executes the generated tests against different web browsers to verify compliance of the latter with the same origin policy. Consider only two web

Table 5.13: Test cases for the history rule

Test string	Oracle predicate
w_{10}	shownVisited(url1) = **false**
w_{11}	shownVisited(url2) = **false**
w_{12}	shownVisited(url4) = **false**
w_{13}	shownVisited(url2) = **false**
w_{14}	shownVisited(url4) = **false**
w_{15}	shownVisited(url4) = **false**
w_{16}	shownVisited(url2) = **false**
w_{17}	shownVisited(url1) = **false**

browsers, Internet Explorer and Firefox. This test execution considers only these browsers because libraries to build the Adaptor and the scripts to automate the execution of test cases against those browsers already exist. Watir (as well as Watij) and FireWatir are libraries that provide scripting control over the behaviour of Internet Explorer and Firefox browsers.

Table 5.14 shows the verdicts produced after the execution of the generated test cases. Internet Explorer fails on test cases tc_1, tc_2 and tc_3 because it shows a cached web page when the *open(url2)* action executes after the *frame(url2)* action opened the *url2* web page from a (different) TLD instance, *url1* and *url3*.

The suite of generated test cases is able to reveal differences between Internet Explorer and Mozilla implementations. It probably follows from different interpretations of the policy. Nevertheless, this fact shows the utility of the generated suite for testing conformance to the policy and, therefore, shows the utility of the testing framework.

Both browsers fail the test cases for the history feature (tc_{10} to tc_{17}). This indicates that both implementations do not comply with the modelled policy. However, it can also indicate an error in the model, that is, that the model does not specify correctly the policy. This is not the case in our study. The privacy model was reviewed and the browsers were subjected to further tests that confirmed the non-compliance with the policy.

5.4.2 Children's Online Privacy Protection Act policy

In this exercise the focus is to show how test cases are generated from a modelled privacy policy that contains *obligations*. The process of executing the generated test cases is similar to the

Table 5.14: Verdicts from the execution of test cases for the same origin policy

Test case	Internet Explorer	Firefox
tc_1	fail	pass
tc_2	fail	pass
tc_3	fail	pass
tc_4	pass	pass
tc_5	pass	pass
tc_6	pass	pass
tc_7	pass	pass
tc_8	pass	pass
tc_9	pass	pass
tc_{10}	fail	fail
tc_{11}	fail	fail
tc_{12}	fail	fail
tc_{13}	fail	fail
tc_{14}	fail	fail
tc_{15}	fail	fail
tc_{16}	fail	fail
tc_{17}	fail	fail

ones described in previous exercises, therefore it is not detailed here. For this exercise, consider a web application that subscribes users for a service. To register a new user the application *collects* certain information about the user. The collection and handling of information in web applications is regulated by different policies and laws. In principle, this application has been designed to comply with the Children's Online Privacy Protection Act (available at `http://www.ftc.gov/ogc/coppa1.htm`) (known as the COPPA policy) that states certain restrictions to the collection of information about children not older than 13 years.

5.4.2.1 Privacy model

The COPPA policy applies to a website operator that pretends to collect individually identifiable information about a child (a user who is aged less than 13). Table 5.15 shows two rules extracted from this policy.

- *Rule 1* states that the website operator can collect *contact information* of the parents of a person aged less than 13 with the sole purpose of requesting parental consent provided that this information is deleted if consent is denied or no consent is granted after some

reasonable waiting time.

- *Rule 2* states that the website operator cannot collect a child's *protected information* unless it has previously received consent from a parent.

Table 5.15: Privacy rules in the COPPA policy

						O		
rule	ρ	a	d	ag	c	β	δ	υ
1	allow	collect	Contact	Website	age $<$ 13 $\wedge$ parent-aut $\neq$ denied	delete	Contact	deny-aut $\vee$ req-time-out
2	deny	collect	Protected	Website	age $<$ 13 $\wedge$ parent-aut $\neq$ granted	-	-	-

Among the *data categories* defined by the policy, two are of interest for this study, the *Contact* category and the *Protected* category, where *Contact* $<$ *Protected*. Additionally, this study considers a unique *user category* in the policy, the *Website* category.

5.4.2.2 Behavioural model

Part of the web application that this exercise studies is modelled in Figure 5.21. This model shows the functionalities that are of interest for testing compliance to the COPPA policy. These functionalities are intended to comply with the policy, however, for test case generation purposes, this model also contains some transitions that violate the policy when executed (marked as arriving to a *fail* state). An application that implements those actions is not complaint with the policy.

The model in Figure 5.21 is described in an abstract way to increase readability. In this model, $q \overset{a(var)}{\rightarrow} q'$ is an abbreviation for $q^i \overset{a(val)}{\rightarrow} q'^i$ for all suitable values val that can be assigned to var. Moreover, an abstract state q in this model represents a set of states q^i. We write q^i to refer to a particular instance of state q.

In the application's model, the action *signIn* represents a user starting the interaction with the application. The user provides a user identification and age into variables *user* and *age* respectively. This action with the variable *user* as a parameter defines a session. In other words, the system deals with several users and for each user it keeps a record with the proper

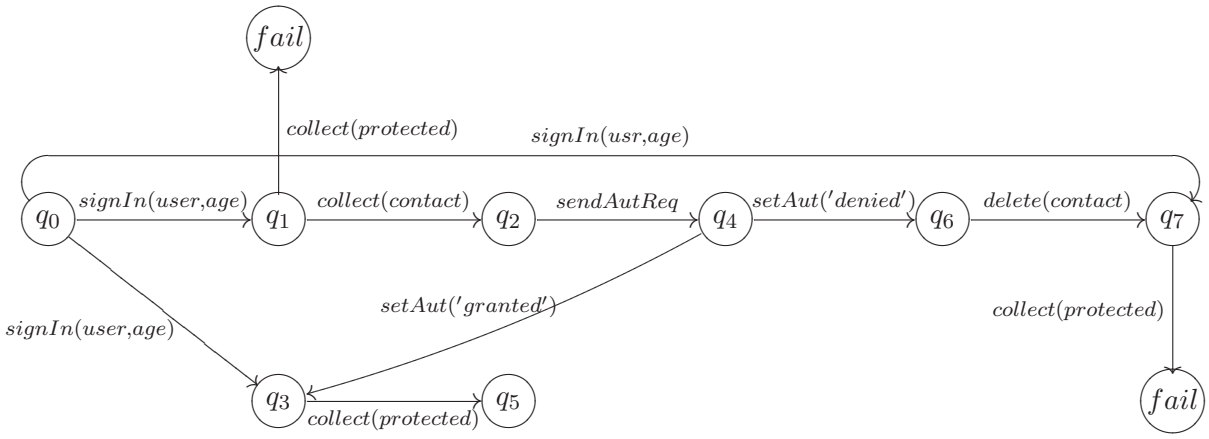


Figure 5.21: Model of a web application that subscribes users for a service.

variables to hold users' information. However, the policy considers the system is dealing with one and only one user at a time, that is, the policy considers that the system has only one set of variables, e.g., one variable *age* and one variable *parent-aut*. Then, the role of the variable *user* is to set the system to use only one from its set of records. Variables *contact* and *protected* are records such that *Contact(contact)* and *Protected(protected)* hold. Finally, this study assumes that the web application stores the data it collects in a repository called *database* and there exists a function *database(u)* that returns a set of the data collected for the particular user u.

Transitions in the abstract representation of the system appear to be non-deterministic. This non-determinism is solved using (state) variables *age* and *parent-aut*. Action *signIn* receives *age* as a parameter and retrieves internally the value of *parent-aut* associated with the value of *user* received as a parameter. The model shows that after executing *signIn* the system's state is

- q_1 if the user's *age* is less than 13 and has no previous record of *parent-aut*,
- q_3 if either the user's *age* is greater than 13 or it is less than 13 and has a granted *parent-aut*,
- q_7 if the user's *age* is less than 13 and has a denied *parent-aut*.

For test generation purposes, actions in the model map to actions in the policy. Sometimes, like in the case of *collect* and *delete* actions, this mapping is done by *name matching*. However,

there are other cases when an explicit mapping relation needs to be defined. In the present study, action *deny-aut* in the policy has no direct mapping to the model. Thus, an explicit mapping is declared between *deny-aut* in the policy and *set-aut(denied)* in the model.

5.4.2.3 Data generation model

An abstract behavioural model is not enough for real systems testing. The test generation in this framework is performed over more specific model transitions. Thus, the testing framework requires the definition of a test scenario for this study. In plain words a scenario refers to the proper instantiation of variables in the abstract model to generate the actual system model.

Consider a testing scenario with four users, *Alice*, *Bob*, *Charles*, and *Denis*. Each user defines its own record of variables {*age, parent-aut*}, as *Alice*={*10, denied*}, *Bob*={*20, nil*}, *Charles*={*12, nil*}, and *Denis*={*11, granted*}. Those records are pre-defined taking into account the conditions in the privacy rule so that each condition can be exercised at least once. Note that there are no limits on the number of users and a larger set of them could be used. However, these four users are enough to generate the test cases.

Then, the framework defines how the user and its associated values are chosen for each test case being generated. The *user* value for a particular test case depends on which condition from the current privacy rule needs to be exercised. Thus, consider global variables *reqAut* and *reqAge* which are updated by the framework when a privacy rule is selected for testing. Consider also global variable *cUser* which records the current user and is updated the first time the value for *user* is generated. The generation of values is guided by the grammars in Figure 5.22.

The condition `[reqAut≠'']` in the grammar for action *setAut* always denies authorisation requests for new users. This is because this study focuses on exploring test cases where the privacy policy is applied and disregards general functional test cases.

5.4.2.4 Test generation

To generate test cases for this study, in the first place, take *rule 1* from the list of rules in Table 5.15. Calculate the rule r by subtracting all previously processed rules, which results

[ ]	[;]	S_{signIn} → USER AGE
[reqAut='denied' and reqAge='<13']	[cUser='Alice']	USER → 'Alice'
[reqAge='>13']	[cUser='Bob']	USER → 'Bob'
[reqAut='' and reqAge='<13']	[cUser='Charles']	USER → 'Charles'
[reqAut='granted' and reqAge='<13']	[cUser='Denis']	USER → 'Denis'
[cUser='Alice']	[;]	AGE → 10
[cUser='Bob']	[;]	AGE → 20
[cUser='Charles']	[;]	AGE → 12
[cUser='Denis']	[;]	AGE → 11

[ ]	[;]	S_{setAut} → AUT
[reqAut≠'']	[;]	AUT → 'granted'
[reqAut='']	[;]	URL → 'denied'

Figure 5.22: Data generation grammars for COPPA policy testing

in the same *rule 1*. Then, select state q_1 from which rule r is applicable. Note that r would be applicable also in q_3 because of the *collect* action. However, $age > 13$ holds in q_3 and this contradicts the condition c making r not applicable. Thus, string w_1 = *signIn('Charles',12)* that reaches q_1 is generated.

Given that r has a ruling $\rho = allow$ and there is an obligation ($O \neq nil$), generate a string wp = *sendAutReq* and append it to w_1. This string enables action υ = *setAut('denied')*. To verify that the obligation is fulfilled generate the predicate p = *contact* $\notin$ *database('Charles')* that must hold in the state after the execution of action β = *delete(contact)*. Then, the test case is generated and added to the test suite. The test string for this test case is *signIn('Charles',12) collect(contact) sendAutReq setAut('denied')*. The behaviour characterised by this string starts at state q_0 and ends in state q_7 where the oracle predicate "contact $\notin$ database(user3)" must hold.

Next, for *rule 2* in Table 5.15, calculate the exceptions set by the first rule. In this case, the rule r gets the d and c fields updated. The resulting rule is shown in Table 5.16.

Then, select states from which r is applicable. States q_1 and q_7 meet that condition. Note that state q_1 has an additional transition with action *collect* and q_3 also has one, however, ¬*Contact(contact)* does not hold in q_1 for this second *collect* transition and condition c does

Table 5.16: Rule generated from the difference of rules $rule2 - rule1$ in Table 5.15

rule	ρ	a	d	ag	c	O β	δ	υ
r	deny	collect	Protected $\wedge$ $\neg$ Contact	Website	age $<$ 13 $\wedge$ parent-aut $=$ denied $\vee$ age $<$ 13 $\wedge$ parent-aut $=$ nil	-	-	-

not hold in q_3, therefore they are not considered. Table 5.17 shows the generated strings that reach states q_1 and q_7, together with the forbidden action and the oracle for the test cases. The oracle uses the boolean function *fail()* that indicates if the system has reached the *fail* state.

Table 5.17: Generated strings for COPPA policy

	String	Denied action	Oracle
w_2	signIn('Alice',10)	collect(protected)	fail()=false
w_3	signIn('Charles',12) collect(contact) sendAutReq() setAut('denied') delete(contact)	collect(protected)	fail()=false

5.4.3 Discussion

The present study focuses on testing compliance with a defined privacy policy. A privacy rule specifies if a defined operation is allowed to be executed or not under a specified condition. The generation of a test sequence that exercises this rule is comparable to the reachability problem. The separation between behavioural, privacy and data models enables the framework to deal with this problem in a tractable way. Firstly, privacy models define explicitly operations that need to be tested. Conditions in the privacy models also restrict the size of the set of variables that need to be considered, reducing in practical terms the size and complexity of the other models. In the second instance, backward and forward search techniques over the behavioural model are used to generate the paths that lead to the execution of the operations defined in the privacy models. Finally, data values can be added to the data model dynamically as the behavioural model drives the exploration of the system.

This study also showed the importance of data models in the generation of concrete test cases. First of all, the behaviour of the system depends on the history of previously selected (or generated) data values, e.g., in the first exercise, pages selected to be opened will be included into the history repository and cached. This means that the definition of the data model restricts the number of enabled actions and behaviours which in its turn impacts the effectiveness of the test generation process. For example, in the first exercise of the study, the grammars defined to generate data did not contemplate, on purpose, cases where two links had to be opened inside a frame consecutively. Therefore, the testing framework generated and executed fifteen (15) test cases out of eighteen (18) required combinations of states and actions.

The predicates in the rules of the data generation models are used in this study to represent searching heuristics. Consider, for example, the grammars in Figure 5.20 where the condition `[!inHistory(X,"one.pps/url1.html")]` restricts the selection of the value `"one.pps/url1.html"`. The first time this value is selected, it is recorded in the *history* respository, thus it will not be selected again. Similar conditions for all rules of the grammar corresponding to the *open* function, restrict the execution of this function to four times, one for each rule. One can argue that this way of restricting the execution of an action is not elegant, but it is effective. Moreover, it serves to illustrate one way of using predicates in the data generation models to guide the selection of values.

5.5 Asynchronous systems testing

This study aims to show the workings of the testing framework in the domain of testing asynchronous systems. The type of asynchronous systems that are of interest for this thesis need to be aware of the occurrence of observable actions. Actions executed by the SUT (and its environment) modify the state of the SUT. By observing the occurrence of these actions, any change in the state of the SUT is reflected on the state of the model. Therefore, observable actions effectively influence the test generation process. Then, this study focuses on the test generation and execution processes.

To deal with observable actions, an on-line approach to test generation and execution is implemented. Automation of this approach is shown and discussed.

Although the modelling of asynchronous systems is not the focus of this study, the SUT and its model are briefly introduced and discussed at the beginning.

5.5.1 The FIX Protocol

Stock monitoring and financial data processing applications are classical examples for asynchronous distributed systems. This case study refers to a trading system that implements the Financial Information eXchange (FIX) protocol[¶]. The FIX protocol defines a series of messaging specifications for the electronic communication of trade-related messages. A trade operation has always a buyer and a seller, represented by a client (that initiates a transaction by buying or selling some security) and a server (which plays the complementary role). Figure 5.23 shows examples of valid sequences of actions for this trading protocol described in a plain notation, similar to the well known Alice Bob notation. The order of the actions in these sequences reflects the point of view of the client.

In Figure 5.23, agents are denoted S for the server and C for the clients. For reading convenience, this description distinguishes between clients that play the role of buyers (C_b) and those who play the role of sellers (C_s). The first sequence shows a client that decides to *sell* X quantity of a defined stock. Then the server acknowledges the request (*sellAck*). Next, the client decides to *cancel* the request and the server can choose the answer by either acknowledging the cancel request (*cancelAck*) or rejecting it (*cancelReject*).

Any system that implements the FIX protocol is asynchronous in the sense that a client can send a message to the server and does not need to wait for the server response to send another one. Consider, for example, the second sequence in Figure 5.23. The client sends a request to *buy* X quantity of a defined stock. Then, the client changes his mind and requests to *cancel* the previous buying order without knowing the answer of the server. The server, in the meantime, rejects the buying order (*buyReject*). Thus, at the time as it processes the cancel order, the server has no other choice than rejecting this order (*cancelReject*) because

[¶]http://www.fixprotocol.org

$$
\begin{array}{lll}
\textbf{(1)} & C_s \to S: & \{sell(1,X)\} \\
 & S \to C_s: & \{sellAck(1)\} \\
 & C_s \to S: & \{cancel(1)\} \\
 & S \to C_s: & \{cancelAck(1,X) + cancelReject(1)\} \\
\textbf{(2)} & C_b \to S: & \{buy(1,X)\} \\
 & C_b \to S: & \{cancel(1)\} \\
 & S \to C_b: & \{buyReject(1)\} \\
 & S \to C_b: & \{cancelReject(1)\} \\
 & & \\
\textbf{(3)} & C_b \to S: & \{buy(1,X)\} \\
 & C_s \to S: & \{sell(2,Y)\} \\
 & C_s \to S: & \{cancel(2)\} \\
 & S \to C_b: & \{buyAck(1)\} \\
 & S \to C_b: & \{(partialFill(1,Y),\ Y < X) + (fullFill(1,X),\ X \leq Y)\} \\
 & S \to C_s: & \{sellAck(2)\} \\
 & S \to C_s: & \{(fullFill(2,Y),\ Y < X) + (partialFill(2,Y),\ X \leq Y)\} \\
 & S \to C_s: & \{(cancelReject(2),\ Y < X) + (cancelAck(2,Y-X),\ X \leq Y)\}
\end{array}
$$

Figure 5.23: Valid sequences of actions in the FIX protocol

the first order (buying order) was never processed or initiated.

In the third sequence in Figure 5.23, two (possibly different) clients send different requests to *buy* and *sell* some X and Y quantity of a defined stock. Immediately, the seller client requests to *cancel* the selling order. In the meantime, the server acknowledges the buying request *buyAck* and processes it. Then the server sends back to the client the response indicating that either the request was completely executed *fullFill* or partially executed *partialFill*. Similarly it sends the acknowledging message and the response to the seller client and then, processes the cancel request. If the selling order was fully executed, then there is no choice for the server other than to reject the cancel request (*cancelReject*). On the other hand, if the selling order was only partially executed then the server acknowledges the cancel request for the remaining of the selling order.

5.5.2 Modelling the FIX Protocol

Financial protocols like FIX usually have large specifications. For purposes of exemplifying the test generation and execution processes from a system's model this case study uses a

simplified description of the FIX protocol . To reduce the complexity of the representation of the model, it uses *actions* with *variables* as parameters. Hereafter, the notation *a(x)* refers to an instance of *a(var)* if x is a value for *var*.

For modelling the FIX protocol, this study uses the concept of a *transaction*. A transaction is identified by the parameter *id* so that all operations with the same *id* belong to the same transaction. Models always refer to a unique transaction. However, the real implementation allows several transactions to be executed concurrently. The complete behaviour of the system is modelled by the parallel composition of as many models as concurrent transactions the system executes.

Consider controllable actions *buy(id,amt)*, *sell(id,amt)*, *cancel(id)* and *alter(id)*. A transaction initiates with actions *buy(id,amt)* or *sell(id,amt)* which place a request for buying or selling, respectively, an amount *amt* of a defined stock. Action *cancel(id)* places a request for cancelling the transaction identified by *id*, and *alter(id)* requests a modification on the transaction identified by *id*.

The server responds by acknowledging the buy/sell request with actions *buyAck(id,amt)* or *sellAck(id,amt)*, or by rejecting the request with actions *buyReject(id)* or *sellReject(id)*. The server also fills a request, partially or completely, with actions *partialFill(id,amt)* or *fullFill(id,amt)* if it receives matching requests (two transactions with a buy and a sell request respectively). Actions *cancelAck(id,amt), cancelReject(id), alterAck(id,amt)* and *alterReject(id)* are suitable responses for the *cancel* and *alter* actions respectively.

A transaction is finalised when an action *buyReject(id)* or *sellReject(id)* is executed by the server. A transaction is also finalised when the request is fullfilled, that is, when action *fullFill(id,amt)* is executed by the server. Any request that refers to a finalised action is rejected, e.g., *cancelReject(id)* is the only possible response for a *cancel(id)* request when the transaction *id* has already finalised.

For illustration purposes only Figure 5.24 shows part of the LTS that describes the model of the FIX implementation. In this model, controllable actions are denoted $\overline{\textbf{sell}}$ and $\overline{\textbf{cancel}}$. Notice the combinations of observable answers of the system that are triggered by only two of the controllable actions.

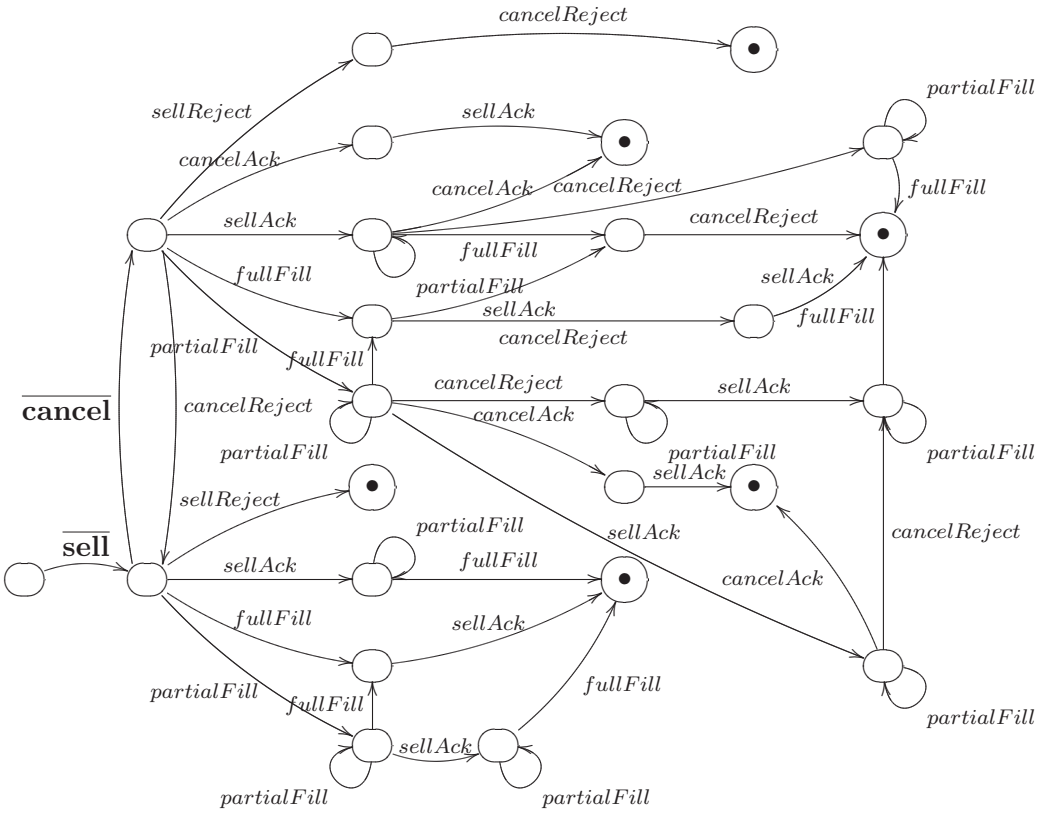


Figure 5.24: Simplified model of the FIX protocol

5.5.3 Test generation and execution

For a practical application, the generation and execution of test cases have specific requirements. On one hand, the test generation process requires the set of possible action executions to be finite, otherwise, the algorithms will run indefinitely. To restrict the size of the set of possible executions, one restricts the number of available *id*s to three. Consider also that the execution process involves communication with other components of the system and requires these components to be available. Additionally, in the real implementation, actions have parameters other than the transaction *id* and the amount *amt* of a defined stock that is traded in the transaction. However, for simplicity, this study does not include these parameters in the discussions.

Consider *QuickFix/J*[‖], an open implementation of the FIX protocol, to provide the required components for the execution of test cases. This implementation has two modules, the

[‖] http://www.quickfixj.org

client module and the server module. Consider also a modified client module that translates generated action messages into appropriate messages to the *server* and that receives messages from the server and translates them to appropriate messages for the testing tool. Figure 5.25 shows an screenshot of this module during the execution of this study.

The present study uses SmartMBT to automate and implement the model-based framework. Figure 5.26 shows a modified SmartMBT tool during the execution of this study. This modified version of the SmartMBT tool includes the ability to observe actions executed by the SUT via the implementation of the Observer module. This module receives a message from the SUT (the modified client module from QuickFix/J) and updates the state of the system in the model.

Figure 5.26 shows actions that have been executed and observed. They are presented as sequential steps of the test. Each one can *pass* if it was expected to happen, or *fail* otherwise. For example, request actions *sell*(1) and *cancel*(1) in steps 2 and 3, which are acknowledged by the server in steps 4 and 6, are correctly processed by the server and receive a *pass* verdict. However, the last step in this sequence forces a *fail* verdict by sending an unexpected action to the observer. It is worthwhile noting also that after step 7 (*sell*(3)), enabled actions are *alter*, *buy*, *cancel*, *init* and *sell*. Nevertheless, enabled *cancel* actions are only *cancel*(2) and *cancel*(3) because *cancel*(1) has been already executed. In the same way, as no more *id*s are available *sell* and *buy* actions are not available any more. Figure 5.25 shows the state of the FIX server which does not need major discussion.

Table 5.18 summarises the workings of the implemented testing process. This table describes the whole process as a sequence of action executions that interleaves execution requests by the testing tool and observation of executions performed by the SUT and the environment. The first column in the table shows actions requested (and executed) by the test generator. The second column shows executed actions observed in the communication channel. Finally, the third column shows the evolution of the set of expected observable actions. This process is executed in a closed environment, so that no other component than a single instance of the testing tool communicates with the server.

Observe in the first row of the table that the generation algorithm chooses to initiate

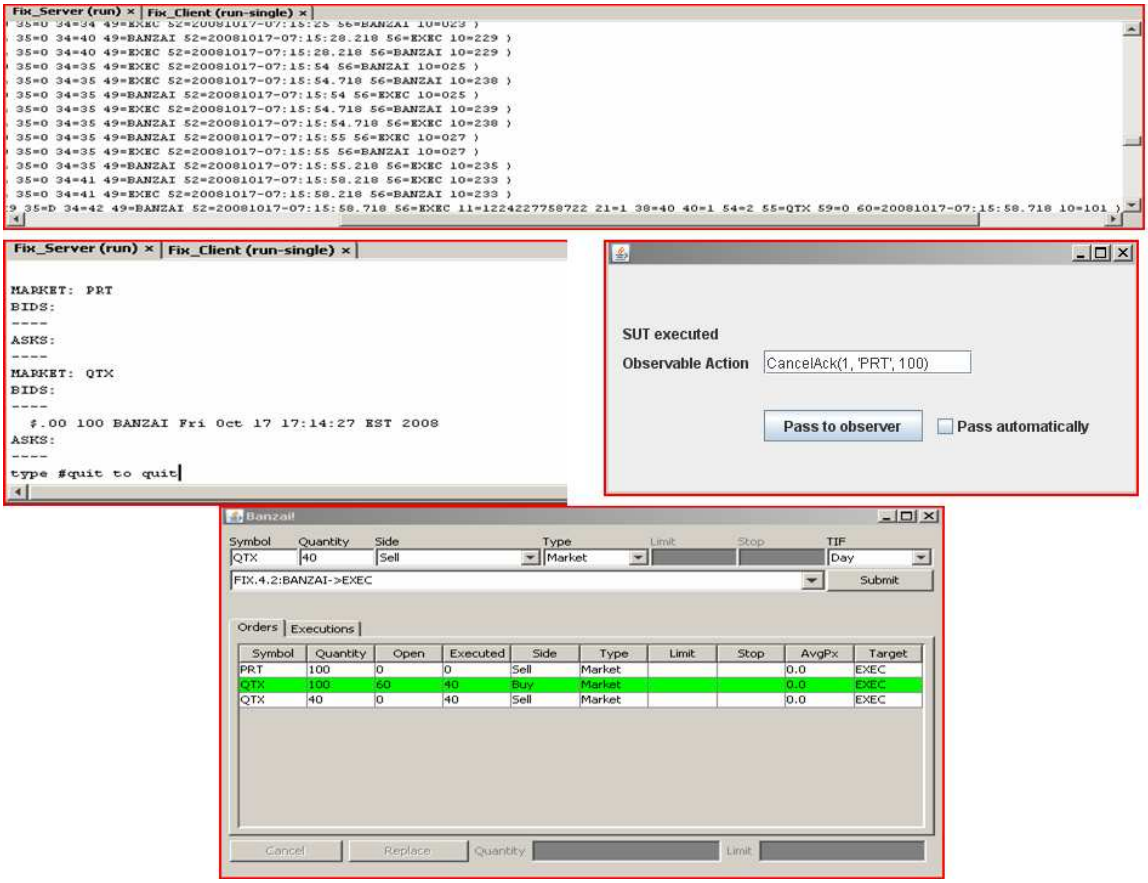


Figure 5.25: Linking to the SUT

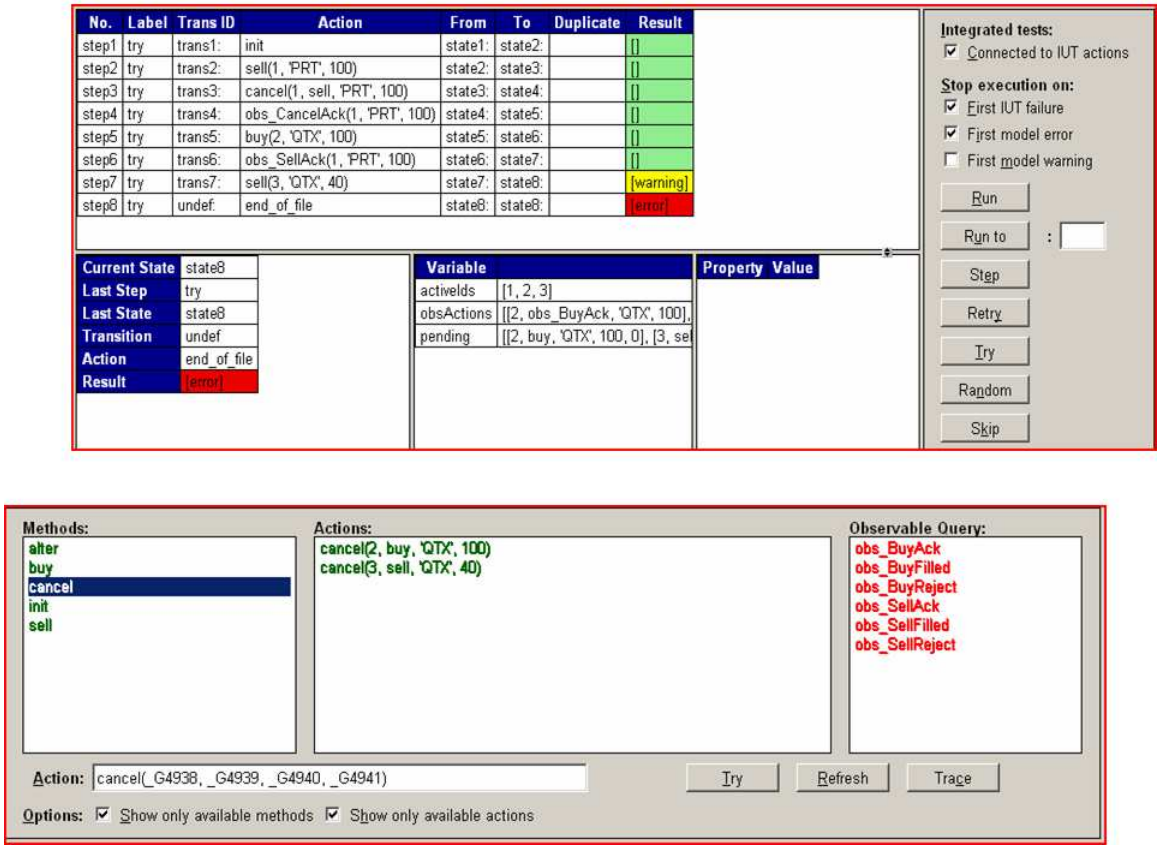


Figure 5.26: The extended SmartMBT

Table 5.18: Evolution of the state of an asynchronous testing process

	Testing Tool	Execution Observer	S_o
1	sell(1,100)		{sellAck(1,100), sellReject(1), fullFill(1,100), partialFill(1,amt<100) }
2	cancel(1)		{sellAck(1,100), sellReject(1), fullFill(1,100), partialFill(1,amt<100), cancelAck(1,amt≤100), cancelReject(1) }
3		sell(1,100) cancel(1)	{sellAck(1,100), sellReject(1), fullFill(1,100), partialFill(1,amt<100), cancelAck(1,amt≤100), cancelReject(1) }
4		cancelAck(1,100)	{sellAck(1,100) }
5	buy(2,100)		{sellAck(1,100), buyAck(2,100), buyReject(2), fullFill(2,100), partialFill(2,amt<100) }
6		sellAck(1,100) buy(2,100)	{buyAck(2,100), buyReject(2), fullFill(2,100), partialFill(2,amt<100) }
7	sell(3,40)		{ buyAck(2,100), buyReject(2), sellAck(3,40), sellReject(3), fullFill(2,100), partialFill(2,amt<100), fullFill(3,40), partialFill(3,amt<40) }
8		buyAck(2) sell(3,40) sellAck(3,40)	{ fullFill(2,100), partialFill(2,amt<100), fullFill(3,40), partialFill(3,amt<40) }
9		partialFill(2,40) fullFill(3,40)	{fullFill(2,60), partialFill(2,amt<60) }
10	cancel(2)		{fullFill(2,60), partialFill(2,amt<60), cancelAck(2,amt≤60), cancelReject(2) }
11		cancel(2) cancelAck(2,60)	{ }

the test case executing action $sell(1, 100)$. Then, the set of expected observable actions is updated (see column S_o). This execution enables actions $cancel(1), alter(1)$ as well as $buy(2, amt), sell(2, amt)$. Note in the second row that action $cancel(1)$ is executed. This updates the set of expected observable actions. Notice also in column S_o that the set in the second row has additional elements $cancelAck(1, 100)$ and $cancelReject(1)$ which correspond to the expected responses for $cancel(1)$.

In the third row of the table, the observer reports actions $sell(1, 100)$ and $cancel(1)$ executed and observed in the communication channel. Internally it does check that those actions are the ones the generator requested and discards them. Then, as the fourth row shows, the observer also receives a message indicating the execution of action $cancelAck(1, 100)$. This execution triggers the update of S_o resulting in $S_o = \{sellAck(1, 100)\}$. Compare it with the previous set of expected observable actions to notice that action $cancelAck(1)$ has been removed from S_o as it has already been executed. Additionally, this execution triggers the removal of events like $sellReject(1)$ as they cannot happen any more.

A similar scheme repeats until row 9 which indicates that the observer notices in the communication channel the execution of $fullFill(3, 40)$ and $partialFill(2, 40)$ by the SUT. This updates the set of expected observable actions by removing any *fill* actions with $id = 3$. This also updates the amount that can be filled for transaction with $id = 2$ to 60.

Finally, in the last two rows there is a request for cancelling the transaction with $id = 2$, which is observed and acknowledged, leaving S_o empty and enabling the termination condition of the algorithms.

The execution history for this study can be read from the column of the Execution Observer module. This sequence of action executions represents a test case. This test case execution produces a *pass* verdict since it does not fail and the algorithms terminate.

5.5.4 Discussion

The model of an asynchronous system is described from the point of view of one of the elements of the system. For different components the definition of which actions are observable and which are controllable varies. In this study, the system is modelled from the point of view of

the *client* component. This component is the SUT and the other components of the system are the environment of the SUT.

The concept of controllable and observable actions is not necessarily linked to separate components. Although the system in this study had two clearly distinct components, a client (the SUT) and a server (the environment), internal actions of the SUT can be considered observable actions. Replacing internal actions with observable actions enables the testing framework to test asynchronously threaded code.

Writing LTS models for asynchronous systems deserves special consideration. A model must include all possible interleavings of controllable and observable actions, a missing interleaving turns out to be a forbidden one. Then, a wrongly specified model becomes unreliable at the time of driving the generation and execution of test cases. Such a model will lead to conflicting verdicts for the same test case. This study showed that even protocols (and systems) with a low number of actions produce large models.

An asynchronous test must be very careful to synchronise with the system that it is observing. This synchronisation is usually performed by locking the test case until a successful change of state is observed or a defined timeout event occurs. The inclusion of observable events in the model enables the test execution process to unblock the test case and continue its execution asynchronously. Observable events create a new state in the SUT where the responses to an executed controllable action are still pending. When the SUT transitions to this state, it unblocks the execution of the test case.

The Adaptor module is responsible for synchronising with the SUT's interface. When a controllable action is selected for execution by the generation algorithm, the Adaptor translates it to the SUT's interface and executes it. The responsibility of the Adaptor ends here. If an error or exception occurs the Adaptor reports a failing test case, otherwise the test case continues to be executed.

The need for two way communication between the model and the SUT (driven by the testing tool) requires more coding effort than the one required for other previously studied domains. This is mainly because the scripts in the Adaptor module need to recognise when an action has been executed by the SUT or its environment. The implementation for the

present study instrumented the code of the SUT.

5.6 Lessons learned

General lessons can be drawn from the experience of executing the different case studies presented in this chapter.

- Models allow to specify precisely which elements or properties of a system are being tested and all conditions and assumptions under which this testing is performed. This precise specification is a necessary condition for assurance. There are domains, such as the security testing domain, for which it is convenient to separate the specification of those elements and properties from the specification of the system's functionalities. This is mainly because these properties are traversal to the functionalities. There are other domains, however, for which the properties to be tested are directly linked to the functionalities and, thus, they do not require specific models for them. Rather, the testing process is documented only from the information obtained from the specification of the functionalities and the tools and techniques used to perform the testing.

- With respect to the models, there is a general architecture for testing security properties, such as the presence of vulnerabilities and conformance to privacy policies. This architecture considers three models. The abstract model represents the high level interface of the SUT. It contains the actions (or operations) that a user can invoke from the application. The implementation model introduces additional properties and functionalities and serves to link the (abstract) behavioural model to the third model. The third model represents the testing objectives. Consider for example the domain of privacy testing where the third model represents the privacy policy. By defining this model in terms of LTS, it can be integrated with the other two models. A privacy policy represents transitions that should, or should not, be enabled in the other models for a defined state, and, in the case of obligations, transitions that should be present on a trace between two given states of the other models.

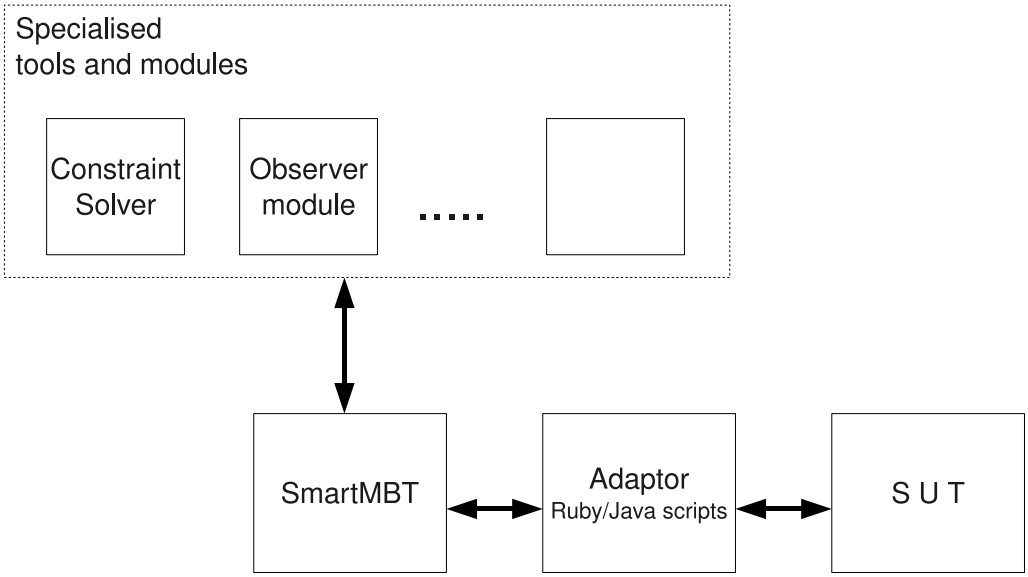


Figure 5.27: Architecture of the implemented model-based framework

- There are other testing domains, such as asynchronous systems and operating systems, for which not three but one model is necessary. The reason is that in these domains the testing objectives are not centred in additional or forbidden functionalities, thus it is not necessary to differentiate between behavioural and implementation models. Moreover, the testing objectives are encoded into specialised algorithms and, in this way, no third model is required.

- There is a general framework based on models that groups the common elements for testing systems in all previously cited domains, security vulnerabilities, privacy, asynchronous systems and operating systems. There are specialised elements that are required for specific domains. These elements are composed with the elements on the general framework to provide specialised frameworks.

- A unique test automation tool that implements the general framework can be used as the central element that drives the testing process over different domains. SmartMBT implements the general framework by allowing the tester to describe transition-based models that represent the behavioural model. Implemented in Prolog, SmartMBT also allows the definition of the data generation models by representing the grammar rules as Prolog rules. The predicates that guard the execution of the rules are also directly included into the Prolog rules. Prolog does not directly allow the execution of actions. However, by declaring the global variables of the system as state variables, SmartMBT allows changes in the valuation of these variables, effectively allowing the transition between global states.

- Specialised domains could require additional tools or techniques to complement the capabilities of a tool such as SmartMBT. For example, in the domain of security vulnerabilities this thesis focused on the generation of successful attacks for a defined vulnerability using a particular operation of the SUT's interface. In the case study for this domain, a customised constraint solver was built to generate the test cases. However, once the test cases were generated, automated execution was not directly possible. Test sequencing was still necessary to bring the system to a state in which these test cases

could be executed. Then, these test cases encoded into the models were used as input to the SmartMBT tool and test sequencing and execution was performed automatically. The architecture of the implemented framework is shown in Figure 5.27.

- The separation between behavioural and data generation models allows concrete models to be built on-the-fly. Concrete states (where state variables have a concrete valuation) are added only when needed, helping to deal with the state explosion problem.

- The use of models provides finer control over the test scenarios. In currently used test automation approaches, such as script-based or keyword-driven approaches, testers define a priori the scenarios in which test cases are executed. The model-based testing approach provides different algorithms that define test scenarios on-the-fly using the information in the models and, in the case of asynchronous systems, the information they get from the observation of the environment. This leads the model-base approach to exercise different test scenarios.

Chapter 6

Conclusions and Future Research Issues

6.1 Thesis summary

This thesis has presented a general model-based framework that can be used for testing a wide range of software systems via diverse specialisations. One of the main characteristics of this general framework is that it maintains a separation between behavioural (or control) models and data generation models. Behavioural models are described using labelled transition systems. Data generation models are described using extended context-free grammars. In these extended grammars, rules are guarded by predicates and contain associated actions. These actions modify the context in which data generation is performed. This context is described as a global state of the system. Actions in the grammars' rules cannot modify the state of the behavioural model. This is a necessary restriction to maintain the integrity of this model. This thesis concludes its presentation of the general framework by describing briefly the process of test generation and test execution.

The general framework requires specialisation to be applied to different testing domains. This thesis describes specialisations of the framework for the domains of security vulnerabilities, privacy policies and asynchronous systems. A specialisation consists of defining an architecture for the models used in test case generation and execution. This architecture de-

fines three separated models for testing vulnerabilities; namely behavioural, implementation and attacker's models; two models required for privacy policies, the behavioural model and the privacy model; and one model for asynchronous systems in which actions are partitioned into controllable and observable actions. Additional specialisations are defined as particular algorithms for test case generation and execution. The generation of test sequences for security vulnerabilities is performed following standard techniques and does not require specialisation. Privacy policies require a specialised test generation algorithm, mainly to make sure that in all possible occurrences of a defined action, the system conforms to a given policy. Asynchronous systems require specialised generation and execution algorithms executed in parallel and following an on-line approach.

This thesis presents four case studies that serve as a demonstration of the applicability of this approach. One of these case studies, which is focused on testing operating systems, shows the general approach applied without further specialisations. The other cases show how the specialised approaches are applied to their respective testing domains.

The case study for the domain of testing vulnerabilities demonstrates how test cases are generated for revealing SQL-injection vulnerabilities. It also explores the modelling of different attacker's objectives, namely, the attacker aiming to get unauthorised access to a system and the attacker aiming to reveal structural information about the design of the system – the structure of the database. This case study also explores the sequencing of actions that include the generated attacks as well as attacks that are performed in more that one step. This last exercise on test sequencing was performed over WebGoat, a web application designed for training security testers that contains several well-known vulnerabilities.

On testing privacy policies, the respective case study explores the modelling of privacy policies using the general structured notation presented in this thesis. Two known policies are used to demonstrate this modelling, the *same origin* policy, defined for preserving the privacy of web browser's users, and the COPPA policy, defined to protect the privacy of children's personal data in web sites that require personal information for user registration purposes. The modelled policies were used to generate test suites for generic implementations of web browsers and web applications. The difference between the two policies is the inclusion of

obligations in the COPPA policy and, therefore, in the generated test cases. As a side result, the generated test cases for the *same origin* policy reveal a faulty implementation of the modelled policy in Internet Explorer, while showing a compliant implementation in Firefox.

In the domain of asynchronous systems, the case study uses a simplified description of the Financial Information eXchange (FIX) protocol to test an open implementation of this standard, the QuickFIX/J application. In this case study, the implementation of a complementary observer module is needed to perform the test generation and execution. This module is implemented in Prolog and connected to the SmartMBT tool.

The case study on device drivers focuses on the applicability of the general framework to an unrelated domain, operating systems. In this case study, the HBA (Host Bus Adapter) storage driver of the OpenSolaris operating system is the target of the testing process. For this particular driver, an automated test suite had been already developed following the functional decomposition approach. This case study demonstrates the use (or reuse) of the elements of this existing test automation framework integrated into the model-based approach. This case study presents explicitly the trade-offs between flexibility in the test case generation process and complexity of the models. More fine-grained actions in the models provide flexibility to the test suites while generating larger models.

Finally, these case studies not only show the applicability of the general and specialised approaches, but also demonstrate that a unique tool can be used as the core element for model-based testing and that required specialisations can be integrated as plug-ins to this tool, or as standalone applications whose outputs are used as inputs by the "core tool". This "core tool" is the SmartMBT tool. Several characteristics make it a suitable choice. For example, its implementation in Prolog allows a direct implementation of the data generation models, and its ability to connect to the SUT allows the use of the on-line approach required for asynchronous testing.

The remainder of this chapter describes how the work performed during the development of the thesis answers the research question established in Chapter 1 and then describes some directions to future and complementary work.

6.2 Answer to the initial research question

A model-based framework for testing security properties contains a behavioural model and a data generation model. The behavioural model provides the control structure for generating and executing sequences of test cases and the data generation model provides the necessary concretisation of the test cases in a way that they can be executed against a real implementation.

Specifically for testing security properties, the behavioural model is logically separated into three models:

- An abstract behavioural model that defines the functionalities of the system that are available through its interfaces.
- An implementation model that defines implementation details of the system. That is, it describes properties of a specific implementation of the system at a level of abstraction lower than the behavioural model.
- A model that represents the testing objectives.

The three models described above are related to each other in specific ways. The implementation models for example, contain description of the internal workings of the actions described in the abstract behavioural models. The third model usually refers to the internal actions described by the implementation models. While the behavioural and implementation models change according to the changes in the SUT, the third model usually changes according to the testing domain.

The behavioural model and the data generation model are sufficient elements to model other kinds of systems, such as asynchronous systems and operating systems. Nevertheless, the process of generating and executing test cases requires some specialisation for these domains.

The implementation of this framework in all of the domains cited above is possible by using existing tools and testing frameworks. Moreover, it is possible to have a unique tool that implements the core of the framework across all of the domains while plugging into it

the elements that provide domain specialisation.

6.3 Future research

This section gives an outlook on possible future work on the topics described in this thesis.

One of the main uses of models in security testing in order to reveal the presence of security vulnerabilities is to specify precisely which kind of vulnerabilities the tester is seeking for. This thesis has presented examples where code-injection vulnerabilities, particularly SQL-injection vulnerabilities, have been modelled in terms of labelled transition systems and associated data generation models. Repositories containing different kinds of vulnerabilities exist to indicate to testing practitioners what to look for when performing penetration testing. Similarly, in order to broaden the use of models in security vulnerability testing practitioners need to be provided with repositories of attack models for several kinds of popular vulnerabilities. This task is quite complex. Experience obtained from the development of this thesis indicates that models cannot be too abstract or too concrete but there exists a right level of abstraction for each kind of vulnerability.

Another interesting research direction resides on the area of search-based software testing [86]. From a theoretical point of view, several searching strategies exist such as simple hill climbing, simulation annealing, estimation of distribution algorithm, or tabu search; however, as Antoniol [7] asserts, it is not clear which ones will be adequate to be applied in, for example, security vulnerabilities testing. A meaningful comparison of these techniques will only be possible if a common set of utilities, tasks and problems is defined with the purpose of performing this comparison. From a practical point of view, automated tools such the ones used in this thesis usually include implementations of searching techniques. SmartMBT and the custom-built constraint solver are armed with implementations of a backtracking searching module. Moreover, the algorithms described in this thesis perform searches and the data generation models, although described in terms of production grammars, can be implemented using a search engine. The implementation of more efficient or specialised searching techniques inside these tools could led to a more cost-effective test automation.

Another important area of improvement concerning the modelling task is to provide practitioners with expressive and adequate languages or notations to describe their models. The approach in this thesis, using LTS, is general enough to allow other languages to be built on top of it. For example, while modelling asynchronous systems our approach assumes that the tester provides a model that describes all possible interleavings of actions. For larger systems this task could be, at least, time-consuming. It would be desirable to have a notation that allows the tester to describe such a model in a succinct way by just defining relationships such as causality (following the ideas of Hendrickson [56]). An automatic translation of such languages into LTS would allow them to be integrated into the approach presented in this thesis.

Finally, the approach presented for testing asynchronous systems can be used to validate the definition of communication protocols. Given two models, the client model and the server model (just to name them), each model can be used to generate test cases to be executed against the other model. This actually should perform a complete simulation of the protocol, assuming that the models are an accurate representation of the protocol's description. The research work in this area will be to provide the theoretical basis to support the premise that such a technique is sound and complete as to provide assurance of the protocol's design. The ideas of Boreale [18] on trace equivalence can provide a starting point.

Appendix A

Acronyms

API	Application Programming Interface
CFG	Context Free Grammars
DNF	Disjunctive Normal Form
FIX	Financial Information eXchange protocol
GUI	Graphical User Interface
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IOTS	Input Output (Labelled) Transition System
LTS	Labelled Transition System
OS	Operating System
PDP	Policy Decision Point
PEP	Policy Enforcement Point
SUT	System Under Test
TLA	Temporal Logic of Actions
XACML	eXtensible Access Control Markup Language

Bibliography

[1] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Hippocratic databases. In *VLDB '2002: Proceedings of the 28th International Conference on Very Large Data Bases*, pages 143–154. VLDB Endowment, 2002.

[2] B. K. Aichernig and P. Pari-Salas. Test Case Generation by OCL Mutation and Constraint Solving. In *Proceedings of the Fifth International Conference on Quality Software*, Melbourne, Australia, September 2005.

[3] B. Anand, H. Krishnankutty, K. Ramakrishnan, and V. Venkatesh. Business rules-based test automation: A novel approach for accelerated testing. In SETLabs Briefings - Infosys, April 2007.

[4] A. H. Anderson. A comparison of two privacy policy languages: EPAL and XACML. In *SWS '06: Proceedings of the 3rd ACM workshop on Secure web services*, pages 53–60, New York, NY, USA, 2006. ACM Press.

[5] R. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. John Wiley and Sons, 1st edition, January 2001.

[6] A. Anton, E. Bertino, N. Li, and T. Yu. A roadmap for comprehensive online privacy policy. Technical Report 2004-47, CERIAS Purdue University, 2004.

[7] G. Antoniol. Search based software testing for software security: Breaking code to make it safer. *Software Testing Verification and Validation Workshop, IEEE International Conference on*, 0:87–100, 2009.

[8] B. Arkin, S. Stender, and G. McGraw. Software penetration testing. *IEEE Security and Privacy*, 03(1):84–87, 2005.

[9] M. Backes, G. Karjoth, W. Bagga, and M. Schunter. Efficient comparison of enterprise privacy policies. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, pages 375–382, New York, NY, USA, 2004. ACM.

[10] M. Barnett, W. Grieskamp, L. Nachmanson, W. Schulte, N. Tillmann, and M. Veanes. Towards a tool environment for model-based testing with AsmL. In A. Petrenko and A. Ulrich, editors, *FATES*, volume 2931 of *Lecture Notes in Computer Science*, pages 252–266. Springer, 2003.

[11] A. Barth, A. Datta, J. C. Mitchell, and H. Nissenbaum. Privacy and contextual integrity: Framework and applications. In *SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P'06)*, pages 184–198, Washington, DC, USA, 2006. IEEE Computer Society.

[12] D. Basin, J. Doser, and T. Lodderstedt. Model driven security: From UML models to access control infrastructures. *ACM Transactions on Software Engineering and Methodology*, 15(1):39–91, 2006.

[13] B. Beizer. *Software Testing Techniques*. Van Norstrand Reinhold Electrical/Computer Science and Engineering Series. Van Nostrand Reinhold Company, New York, 2nd edition, 1990.

[14] A. Belinfante, J. Feenstra, R. de Vries, J. Tretmans, N. Goga, L. Feijs, S. Mauw, and L. Heerink. Formal test automation: A simple experiment. In G. Csopaki, S. Dibuz, and K. Tarnay, editors, 12^{th} *Int. Workshop on Testing of Communicating Systems*, pages 179–196. Kluwer Academic Publishers, 1999.

[15] A. Bertolino. Software testing research: Achievements, challenges, dreams. In *FOSE '07: 2007 Future of Software Engineering*, pages 85–103, Washington, DC, USA, 2007. IEEE Computer Society.

[16] P. Bhateja, P. Gastin, and M. Mukund. A fresh look at testing for asynchronous communication. In *Automated Technology for Verification and Analysis*, number 4218 in Lecture Notes in Computer Science, pages 369 – 383. Springer, 2006.

[17] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks*, 14:25–59, 1987.

[18] M. Boreale, R. D. Nicola, and R. Pugliese. Trace and testing equivalence on asynchronous processes. *Inf. Comput.*, 172(2):139–164, 2002.

[19] M. Bozga, J.-C. Fernandez, L. Ghirvu, S. Graf, J.-P. Krimm, L. Mounier, and J. Sifakis. IF: An intermediate representation for SDL and its applications. In *SDL Forum*, pages 423–440, 1999.

[20] S. Bracher and P. Krishnan. Enabling Security Testing from Specification to Code. In *Fifth International Conference on Integrated Formal Methods (IFM)*, volume 3771 of *LNCS*, pages 150–166. Springer Verlag, November 2005.

[21] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM*, 31(3):560–599, 1984.

[22] P. Brucker. The Chinese Postman problem for mixed graphs. In H. Noltemeier, editor, *WG*, volume 100 of *Lecture Notes in Computer Science*, pages 354–366. Springer, 1980.

[23] J. Bryans. Reasoning about XACML policies using CSP. In E. Damiani and H. Maruyama, editors, *Proceedings of the 2nd ACM Workshop On Secure Web Services, SWS 2005*, pages 28–35. ACM, 2005.

[24] J. Bryans and J. S. Fitzgerald. Formal engineering of XACML access control policies in VDM++. In M. Butler, M. G. Hinchey, and M. M. Larrondo-Petrie, editors, *ICFEM*, volume 4789 of *Lecture Notes in Computer Science*, pages 37–56. Springer, 2007.

[25] C. J. Budnik, R. Subramanyan, and M. Vieira. Industrial requirements to benefit from test automation tools for GUI testing. In R. Koschke, O. Herzog, K.-H. Rdiger, and

M. Ronthaler, editors, *GI Jahrestagung (2)*, volume 110 of *LNI*, pages 422–430. GI, September 2007.

[26] H. Buwalda. Getting automated testing under control. *Software Testing and Quality Engineering*, pages 39–44, November/December 1999.

[27] H. Buwalda. Action figures. *Software Testing and Quality Engineering*, pages 42–47, March/April 2003.

[28] C. Campbell, W. Grieskamp, L. Nachmanson, W. Schulte, N. Tillmann, and M. Veanes. Testing concurrent object-oriented systems with Spec Explorer. In J. Fitzgerald, I. J. Hayes, and A. Tarlecki, editors, *FM*, volume 3582 of *Lecture Notes in Computer Science*, pages 542–547. Springer, 2005.

[29] G. Canfora, C. A. Visaggio, and V. Paradiso. A test framework for assessing effectiveness of the data privacy policy's implementation into relational databases. In *Proceedings of the International Conference on Availability, Reliability and Security*, pages 240–247, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[30] R. Castanet, O. Koné, and P. Laurençot. On the fly test generation for real time protocols. In *ICCCN*, pages 378–387. IEEE, 1998.

[31] I. Castellani and M. Hennessy. Testing theories for asynchronous languages. In *Proceedings of 18th International Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS'98*, number 1530 in Lecture Notes in Computer Science, pages 90–101. Springer, 1998.

[32] R. Chinchani, A. Iyer, H. N. Q., and S. Upadhyaya. A target-centric formal model for insider threat and more. Technical Report 16, Department of Computer Science and Engineering, University at Buffalo, 2004.

[33] T. S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, 4(3):178–187, 1978.

[34] Cigital Inc. and MITRE Corporation. "The Common Attack Pattern Enumeration and Classification (CAPEC) Initiative". http://capec.mitre.org.

[35] E. M. Clarke, S. Jha, and W. Marrero. Verifying security protocols with Brutus. *ACM Transactions on Software Engineering and Methodology*, 9(4):443–487, 2000.

[36] R. Cleaveland and S. A. Smolka. Strategic directions in concurrency research. *ACM Computing Surveys*, 28(4):607–625, 1996.

[37] L. Cranor, B. Dobbs, S. Egelman, G. Hogben, J. Humphrey, M. Langheinrich, M. Marchiori, M. Presler-Marshall, J. M. Reagle, M. Schunter, D. A. Stampley, and R. Wenning. The Platform for Privacy Preferences 1.1 (P3P1.1) Specification. Note NOTE-P3P11-20061113, World Wide Web Consortium (W3C), November 2006.

[38] R. de Vries. Towards formal test purposes. In G. Tretmans and H. Brinksma, editors, *Formal Approaches to Testing of Software 2001 (FATES'01)*, volume NS-01-4 of *BRICS Notes Series*, pages 61–76, Aarhus, Denkmark, August 2001.

[39] A. C. Dias Neto, R. Subramanyan, M. Vieira, and G. H. Travassos. A survey on model-based testing approaches: a systematic review. In *Proceedings of the 1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies*, pages 31–36, New York, NY, USA, 2007. ACM.

[40] W. Du and A. P. Mathur. Categorization of software errors that led to security breaches. In *Proc. 21st NIST-NCSC National Information Systems Security Conference*, pages 392–407, 1998.

[41] W. Du and A. P. Mathur. Vulnerability testing of software system using fault injection. Technical report, COAST, Purdue University, West Lafayette, IN, US, Apr. 1998.

[42] I. K. El-Far and J. A. Whittaker. Model-based software testing. *Encyclopedia on Software Engineering*, 2001.

[43] M. Fewster and D. Graham. *Software test automation: effective use of test execution tools*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1999.

[44] W. L. Fithen, S. V. Hernan, P. F. ORourke, and D. A. Shinberg. Formal modeling of vulnerability. *Bell Labs Technical Journal*, 8(4):173–186, 2004.

[45] T. Frühwirth. Theory and Practice of Constraint Handling Rules. *The Journal of Logic Programming*, 1994.

[46] D. Gelperin and B. Hetzel. The growth of software testing. *Communications of the ACM*, 31(6):687–695, 1988.

[47] J. Grabowski, D. Hogrefe, G. Réthy, I. Schieferdecker, A. Wiles, and C. Willcock. An introduction to the testing and test control notation (TTCN-3). *Computer Networks*, 42(3):375–403, 2003.

[48] W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes. Generating finite state machines from abstract state machines. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 112–122, 2002.

[49] W. Grieskamp, L. Nachmanson, N. Tillmann, and M. Veanes. Test case generation from AsmL specifications. In E. Börger, A. Gargantini, and E. Riccobene, editors, *Abstract State Machines*, volume 2589 of *Lecture Notes in Computer Science*, page 413. Springer, 2003.

[50] R. G. Hamlet. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, 3(4):279–290, 1977.

[51] A. Hartman, M. Katara, and S. Olvovsky. Choosing a test modeling language: A survey. In *HVC 06: Second International Haifa Verification Conference*, volume 4383, pages 204–218. Springer, 2007.

[52] L. Hayes. *The Automated Testing Handbook*. Software Testing Institute, 1996.

[53] L. Hayes. Evolution of automated software testing. *Automated Software Testing Magazine, August*, pages 14–20, 2009.

[54] J. He and C. A. R. Hoare. Linking theories in probabilistic programming. *Information Sciences*, 119(3-4):205–218, 1999.

[55] G. Helmer, J. Wong, M. Slagell, V. Honavar, L. Miller, and R. Lutz. Software fault tree and colored Petri net based specification, 2000. Online: http://citeseer.ist.psu.edu/article/helmer02software.html.

[56] S. A. Hendrickson, E. M. Dashofy, A. Bhor, R. N. Taylor, S. Li, and N. Nguyen. An approach for tracing and understanding asynchronous systems. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering*, 2002.

[57] O. Henniger. On test case generation from asynchronously communicating state machines. In *Proceedings of the 10th International Workshop on Testing of Communicating Systems*, Cheju Island, South Korea, 1997.

[58] F. Herbreteau, G. Sutre, and T. Q. Tran. Unfolding concurrent well-structured transition systems. In O. Grumberg and M. Huth, editors, *Proc. of the 13th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS07)*, volume 4424 of *Lecture Notes in Computer Science*, pages 706–720, Braga, Portugal, mar 2007. ETAPS, Springer.

[59] R. Hierons, K. Bogdanov, J. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, G. Luettgen, A. Simons, S. Vilkomir, M. Woodward, and H. Zedan. Working together: Formal methods and testing. *ACM Computing Surveys*, 2006.

[60] R. M. Hierons, K. Bogdanov, J. P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, G. Lüttgen, A. J. H. Simons, S. Vilkomir, M. R. Woodward, and H. Zedan. Using formal specifications to support testing. *ACM Computing Surveys*, 41(2):1–76, 2009.

[61] M. Hilty, D. Basin, and A. Pretschner. On obligations. In S. de Capitani di Vimercati, P. Syverson, and D. Gollmann, editors, *10th European Symposium on Research in Computer Security (ESORICS 2005)*, volume 3679 of *LNCS*, pages 98–117. Springer-Verlag, 2005.

[62] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[63] V. C. Hu, E. Martin, J. Hwang, and T. Xie. Conformance checking of access control policies specified in XACML. In *COMPSAC '07: Proceedings of the 31st Annual International Computer Software and Applications Conference - Vol. 2- (COMPSAC 2007)*, pages 275–280, Washington, DC, USA, 2007. IEEE Computer Society.

[64] J. Hui, L. Yuqing, L. Pei, G. Shuhang, and G. Jing. LKDT: A Keyword-Driven Based Distributed Test Framework. *Computer Science and Software Engineering, International Conference on*, 2:719–722, 2008.

[65] K. Irwin, T. Yu, and W. H. Winsborough. On the modeling and analysis of obligations. In *CCS '06: Proceedings of the 13th ACM conference on Computer and communications security*, pages 134–143, New York, NY, USA, 2006. ACM Press.

[66] C. Jackson, A. Bortz, D. Boneh, and J. C. Mitchell. Protecting browser state from web privacy attacks. In *WWW '06: Proceedings of the 15th international conference on World Wide Web*, pages 737–744, New York, NY, USA, 2006. ACM Press.

[67] C. Jard and T. Jéron. TGV: theory, principles and algorithms: A tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. *International Journal on Software Tools for Technology Transfer*, 7(4):297–315, 2005.

[68] D. Johnson. Evolution of automated software testing. *Automated Software Testing Magazine, August*, pages 28–36, 2009.

[69] G. Karjoth and M. Schunter. A privacy policy model for enterprises. In *CSFW '02: Proceedings of the 15th IEEE workshop on Computer Security Foundations*, page 271, Washington, DC, USA, 2002. IEEE Computer Society.

[70] M. Kim, S. Hong, C. Hong, and T. Kim. Model-based kernel testing for concurrency bugs through counter example replay. In *MBT'09, 5th Int. Workshop on Model-Based Testing, (co-located with ETAPS'2009)*, ENTCS, York, United Kingdom, Mar. 2009. To appear.

[71] D. R. Kuhn. Fault classes and error detection capability of specification-based testing. *ACM Transactions on Software Engineering and Methodology*, 8(4):411–424, October 1999.

[72] D. R. Kuhn, R. Chandramouly, and R. W. Butler. Cost effctive use of formal methods in verification and validation. Foundations '02 Workshop, US Dept of Defense, October 2002. Internet: http://csrc.nist.gov/staff/kuhn/kuhn-chandramouli-butler-02.pdf.

[73] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.

[74] C. E. Landwehr, A. R. Bull, J. P. McDermott, and W. S. Choi. A taxonomy of computer program security flaws. *ACM Computing Surveys*, 26(3):211–254, 1994.

[75] P. Larson. Testing Linux with Linux Test Project. In *Proceedings of the 2002 Ottawa Linux Symposium*, Ottawa, Canada, July 2002.

[76] G. Laycock. *The Theory and Practice of Specification Based Software Testing.* PhD thesis, University of Sheffield, 1993.

[77] B. W. Long, C. J. Fidge, and A. Cerone. A z based approach to verifying security protocols. In *ICFEM*, pages 375–395, 2003.

[78] E. Martin. Automated test generation for access control policies. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 752–753, New York, NY, USA, 2006. ACM.

[79] E. Martin. Testing and analysis of access control policies. *International Conference on Software Engineering Companion*, 0:75–76, 2007.

[80] E. Martin and T. Xie. A fault model and mutation testing of access control policies. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 667–676, New York, NY, USA, 2007. ACM Press.

[81] E. Martin, T. Xie, and T. Yu. Defining and measuring policy coverage in testing access control policies. In *Proceedings of the 8th International Conference on Information and*

Communications Security, volume 4307 of *Lecture Notes in Computer Science*, pages 139–158. Springer, 2006.

[82] A. Masood, R. Bhatti, A. Ghafoor, and A. Mathur. Model-based testing of access control systems that employ RBAC policies. Technical report, School of Electrical & Computer Engineering, Purdue University, 2005.

[83] A. Masood, R. Bhatti, A. Ghafoor, and A. Mathur. Scalable and effective test generation for role based access control systems. Technical report, School of Electrical & Computer Engineering, Purdue University, 2006.

[84] J. P. McDermott. Attack net penetration testing. In *NSPW '00: Proceedings of the 2000 workshop on New security paradigms*, pages 15–21, New York, NY, USA, 2000. ACM Press.

[85] G. McGraw. Software security. *IEEE Security and Privacy*, 02(2):80–83, 2004.

[86] P. McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14(2):105–156, 2004. Department of Computer Science, University of Sheffield, Regent Court, 211 Portobello Street, Sheffield S1 4DP, U.K.

[87] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

[88] MITRE Corporation. "The Common Weakness Enumeration (CWE) Initiative". http://cwe.mitre.org/.

[89] L. J. Morell. A theory of fault-based testing. *IEEE Transactions on Software Engineering*, 16(8):844–857, August 1990.

[90] T. Mouelhi, F. Fleurey, B. Baudry, and Y. Traon. A model-based framework for security policy specification, deployment and testing. In *MoDELS '08: Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems*, pages 537–552, Berlin, Heidelberg, 2008. Springer-Verlag.

[91] T. Mouelhi, Y. L. Traon, and B. Baudry. Transforming and selecting functional test cases for security policy testing. *Software Testing, Verification, and Validation, 2008 International Conference on*, 0:171–180, 2009.

[92] L. Nachmanson, M. Veanes, W. Schulte, N. Tillmann, and W. Grieskamp. Optimal strategies for testing nondeterministic systems. In G. S. Avrunin and G. Rothermel, editors, *ISSTA*, pages 55–64. ACM, 2004.

[93] S. Naito and M. Tsunoyama. Fault detection for sequential machines by transition tours. In *Proceedings of 11th IEEE Fault Tolerant Computing Symposium*, pages 238–243, 1981.

[94] A. Nakata, T. Higashino, and K. Taniguchi. Protocol synthesis from context-free processes using event structures. In *Proc. of the 5th International Conference on Real-Time Computing Systems and Applications*, pages 173–180, Hiroshima, Japan, October 1998.

[95] M. Nielsen, G. D. Plotkin, and G. Winskel. Petri nets, event structures and domains, part I. *Theoretical Computer Science*, 13:85–108, 1981.

[96] A. Offutt and S. Liu. Generating test data from SOFL specifications. *The Journal of Systems and Software*, 49(1):49–62, December 1999.

[97] OWASP Foundation. OWASP Testing Guide v 3.0, December 2008. Available online http://www.owasp.org/images/5/56/OWASP_Testing_Guide_v3.pdf.

[98] P. Pari and B. K. Aichernig. Automatic test case generation for OCL: A mutation approach. Technical Report 321, United Nations University - International Institute for Software Technology (UNU-IIST), Macau SAR, China, 2005.

[99] A. Petrenko. Fault model-driven test derivation from finite state models: Annotated bibliography. In *Proceedings of the MOdelling and VErifying parallel Processes - MOVEP*, pages 196–205, 2000.

[100] C. Phillips and L. P. Swiler. A graph-based system for network-vulnerability analysis. In *NSPW '98: Proceedings of the 1998 workshop on New security paradigms*, pages 71–79, New York, NY, USA, 1998. ACM Press.

[101] B. Potter and G. McGraw. Software security. *IEEE Security and Privacy*, 02(2):80–83, 2004.

[102] C. Powers and M. Schunter. Enterprise privacy authorization language (EPAL 1.2). W3C Member Submission, November 2003.

[103] A. Pretschner. Model-based testing. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 722–723, New York, NY, USA, 2005. ACM Press.

[104] A. Pretschner, H. Lotzbeyer, and J. Philipps. Model based testing in evolutionary software development. In *Proc. of the 12th IEEE Intl. Workshop on Rapid System Prototyping (RSP'01)*, pages 155–161, Monterey, California, June 2001.

[105] A. Pretschner, T. Mouelhi, and Y. L. Traon. Model-based tests for access control policies. In *ICST '08: Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, pages 338–347, Washington, DC, USA, 2008. IEEE Computer Society.

[106] C. Ramakrishnan and R. Sekar. Model-based vulnerability analysis of computer systems. In *Second International Workshop on Verification, Model Checking, and Abstract Interpretation (VMCAI'98; Pisa*, sep 1998.

[107] C. Reade and P. Froome. *Software Reliability Handbook*, chapter Formal Methods for Reliability, pages 51–82. Elsevier, March 1990.

[108] H. Robinson. Intelligent test automation. Software Testing and Quality Engineering magazine, October 2000.

[109] R. C. Seacord and A. Householder. A structured approach to classifying security vulnerabilities. Technical Report CMU/SEI-2005-TN-003, Carnegie Mellon University - Software Engineering Institute, 2005.

[110] O. Sheyner, J. Haines, S. Jha, R. Lippmann, and J. M. Wing. Automated generation and analysis of attack graphs. *sp*, 00:273, 2002.

[111] E. W. Stark. Connections between a concrete and an abstract model of concurrent systems. In *In Fifth Conference on the Mathematical Foundations of Programming Semantics, Springer-Verlag. Lecture Notes in Computer Science*, pages 53–79. Springer, 1989.

[112] P. Strooper and L. Wildman. Testing concurrent Java components. In *ICSE COMPANION '07: Companion to the proceedings of the 29th International Conference on Software Engineering*, pages 161–162, Washington, DC, USA, 2007. IEEE Computer Society.

[113] T. Sugeta, J. C. Maldonado, and W. E. Wong. Mutation testing applied to validate SDL specifications. In *Testing of Comunicating Systems: 16th IFIP International Conference, TestCom 2004*, volume 2978 of *LNCS*, pages 193–208, March 2004.

[114] F. Swiderski and W. Snyder. *Threat Modeling*. Microsoft Press, 2004.

[115] H. H. Thompson. Why security testing is hard. *IEEE Security and Privacy*, 01(4):83–86, 2003.

[116] G. Tretmans and A. Belinfante. Automatic testing with formal methods. Technical Report TR-CTIT-99-17, Centre for Telematics and Information Technology, University of Twente, December 1999.

[117] J. Tretmans. Model based testing with labelled transition systems.

[118] J. Tretmans. *A Formal Approach to Conformance Testing*. PhD thesis, University of Twente, Enschede, The Netherlands, 1992.

[119] J. Tretmans. *Formal Methods and Testing*, volume 4949 of *Lecture Notes in Computer Science*, chapter Model Based Testing with Labelled Transition Systems, pages 1–38. Springer Berlin Heidelberg, 2008.

[120] S. Türpe. Security testing: Turning practice into theory. *Software Testing Verification and Validation Workshop, IEEE International Conference on*, 0:295–302, 2008.

[121] M. Utting. Position paper: Model based testing, 2005. Online: http://vstte.ethz.ch/papers.html.

[122] M. Utting, A. Pretschner, and B. Legeard. A taxonomy of model-based testing. Technical Report 04/2006, Department of Computer Science, The University of Waikato, New Zealand, April 2006.

[123] R. van Glabbeek and G. Plotkin. Event structures for resolvable conflicts. In *Proceedings of the 29th International Symposium on Mathematical Foundation of Computer Science (MFCS04)*, volume 3153 of *Lectures Notes in Computer Science*, page 550. Springer, 2004.

[124] M. Veanes, C. Campbell, W. Schulte, and P. Kohli. On-the-fly testing of reactive systems. Technical Report MSR-TR-2005-05, Microsoft Research, 2005.

[125] J. Viega and G. McGraw. *Building Secure Software: How to Avoid Security Problems the Right Way (Addison-Wesley Professional Computing Series)*. Addison-Wesley Professional, October 2001.

[126] J. Warmer and A. Kleppe. *The Object Constraint Language Second Edition: Getting your models ready for MDA*. Adisson-Wesley, 2003.

[127] P. Wegner. Why interaction is more powerful than algorithms. *Communications of the ACM*, 40(5):80–91, 1997.

[128] J. A. Whittaker and M. G. Thomason. A Markov chain model for statistical software testing. *IEEE Transactions on Software Engineering*, 20(10):812–824, 1994.

[129] G. Wimmel. *Model-based Development of Security-Critical Systems*. PhD thesis, Technical University of Munich, 2005.

[130] G. Winskel. Event structure semantics for CCS and related languages. In *Proceedings of the 9th Colloquium on Automata, Languages and Programming*, pages 561–576, London, UK, 1982. Springer-Verlag.

[131] G. Winskel and M. Nielsen. Models for concurrency. In *Handbook of Logic in Computer Science*, pages 1–148. Oxford University Press, 1995.

[132] XACML Technical Committee. eXtensible Access Control Markup Language (XACML) Version 2.0, February 2005.

[133] D. Xu and K. E. Nygard. Threat-driven modeling and verification of secure software using aspect-oriented petri nets. *IEEE Transactions on Software Engineering*, 32(4):265–278, 2006.